



AT&T

307-225
Issue 1

UNIX™ System V Release 3

Programmer's Guide

Amazon Ex. 1010

IPR Petition - USP 7,784,058

Oracle Ex. 1010

Oracle v. VirtaMove, IPR2025-00966

©1986 AT&T
All Rights Reserved
Printed in USA

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

PDP is a trademark of Digital Equipment.
Teletype is a registered trademark of AT&T.
UNIX is a trademark of AT&T.
VT100 is a trademark of Digital Equipment.

Table of Contents

Introduction	xiii
Chapter 1: Programming in a UNIX System Environment: An Overview	
Introduction	1-1
UNIX System Tools and Where You Can Read About Them	1-4
Three Programming Environments	1-7
Summary	1-9
Chapter 2: Programming Basics	
Introduction	2-1
Choosing a Programming Language	2-2
After Your Code Is Written	2-9
The Interface Between a Programming Language and the UNIX System	2-14
Analysis/Debugging	2-50
Program Organizing Utilities	2-74
Chapter 3: Application Programming	
Introduction	3-1
Application Programming	3-2
Language Selection	3-5

Advanced Programming Tools	3-13
Programming Support Tools	3-22
Project Control Tools	3-36
liber , A Library System	3-40

Chapter 4: **awk**

Introduction	4-1
Using awk	4-17
Input and Output	4-18
Patterns	4-29
Actions	4-35
Special Features	4-42

Chapter 5: **lex**

An Overview of lex Programming	5-1
Writing lex Programs	5-3
Running lex under the UNIX System	5-18

Chapter 6: **yacc**

Introduction	6-1
Basic Specifications	6-4
Parser Operation	6-13
Ambiguity and Conflicts	6-18
Precedence	6-24

Error Handling	6-28
The yacc Environment	6-32
Hints for Preparing Specifications	6-34
Advanced Topics	6-38
Examples	6-45
Chapter 7: File and Record Locking	
Introduction	7-1
Terminology	7-2
File Protection	7-4
Selecting Advisory or Mandatory Locking	7-18
Chapter 8: Shared Libraries	
Introduction	8-1
Using a Shared Library	8-2
Building a Shared Library	8-14
Summary	8-46
Chapter 9: Interprocess Communication	
Introduction	9-1
Messages	9-2
Semaphores	9-40
Shared Memory	9-78

Chapter 10: **curses/terminfo**

Introduction	10-1
Overview	10-3
Working with curses Routines	10-9
Working with terminfo Routines	10-64
Working with the terminfo Database	10-70
curses Program Examples	10-83

Chapter 11: The Common Object File Format (COFF)

Chapter 12: The Link Editor

The Link Editor	12-1
Link Editor Command Language	12-4
Notes and Special Considerations	12-22
Syntax Diagram for Input Directives	12-32

Chapter 13: **make**

Introduction	13-1
Basic Features	13-2
Description Files and Substitutions	13-7
Recursive Makefiles	13-12
Source Code Control System Filenames: the Tilde	13-18
Command Usage	13-22

Suggestions and Warnings	13-25
Internal Rules	13-26
Chapter 14: Source Code Control System (SCCS)	
Introduction	14-1
SCCS For Beginners	14-2
Delta Numbering	14-7
SCCS Command Conventions	14-10
SCCS Commands	14-12
SCCS Files	14-39
Chapter 15: sdb —The Symbolic Debugger	
Introduction	15-1
Using sdb	15-2
Chapter 16: lint	
Introduction	16-1
Usage	16-2
lint Message Types	16-4
Chapter 17: C Language	
Introduction	17-1
Lexical Conventions	17-2
Storage Class and Type	17-6

Operator Conversions	17-9
Expressions and Operators	17-12
Declarations	17-24
Statements	17-38
External Definitions	17-44
Scope Rules	17-46
Compiler Control Lines	17-48
Types Revisited	17-52
Constant Expressions	17-57
Portability Considerations	17-58
Syntax Summary	17-59
Appendix A: Index to Utilities	
Glossary	
Index	

List of Figures

Figure 2-1: Using Command Line Arguments to Set Flags	2-16
Figure 2-2: Using argv[n] Pointers to Pass a Filename	2-17
Figure 2-3: C Language Standard I/O Subroutines	2-20
Figure 2-4: String Operations	2-22
Figure 2-5: Classifying ASCII Character-Coded Integer Values	2-24
Figure 2-6: Conversion Functions and Macros	2-26
Figure 2-7: Manual Page for gets(3S)	2-28
Figure 2-8: How gets Is Used in a Program	2-30
Figure 2-9: A Version of stdio.h	2-31
Figure 2-10: Environment and Status System Calls	2-39
Figure 2-11: Process Status	2-40
Figure 2-12: Example of fork	2-44
Figure 2-13: Example of a popen pipe	2-46
Figure 2-14: Signal Numbers Defined in /usr/include/sys/signal.h	2-48
Figure 2-15: Source Code for Sample Program	2-51
Figure 2-16: cflow Output, No Options	2-55
Figure 2-17: cflow Output, Using -r Option	2-56
Figure 2-18: cflow Output, Using -ix Option	2-57
Figure 2-19: cflow Output, Using -r and -ix Options	2-58
Figure 2-20: ctrace Output	2-60
Figure 2-21: cxref Output, Using -c Option	2-64
Figure 2-22: lint Output	2-69
Figure 2-23: prof Output	2-71
Figure 2-24: make Description File	2-75
Figure 2-25: nm Output, with -f Option	2-78
 Figure 3-1: The fcntl.h Header File	 3-16
 Figure 4-1: awk Keywords	 4-3
Figure 4-2: awk Assignment Operators	4-4
Figure 4-3: awk Arithmetic Operators	4-5
Figure 4-4: awk Relational Operators	4-5
Figure 4-5: awk Logical Operators	4-6
Figure 4-6: Operators for Matching Regular Expressions in awk	4-6
Figure 4-7: Numeric Values for String Constants	4-9
Figure 4-8: String Values for String Constants	4-9
Figure 4-9: Built-in Functions for Arithmetic and String Operations	4-11
Figure 4-10: awk String Functions	4-12
Figure 4-11: Sample Input File, countries	4-20

Figure 5-1: Creation and Use of a Lexical Analyzer with lex	5-2
Figure 8-1: a.out Files Created Using an Archive Library and a Shared Library	8-8
Figure 8-2: Processes Using an Archive and a Shared Library	8-9
Figure 8-3: A Branch Table in a Shared Library	8-12
Figure 8-4: Imported Symbols in a Shared Library	8-33
Figure 9-1: ipc_perm Data Structure	9-6
Figure 9-2: Operation Permissions Codes	9-10
Figure 9-3: Control Commands (Flags)	9-10
Figure 9-4: msgget() System Call Example	9-15
Figure 9-5: msgctl() System Call Example	9-22
Figure 9-6: msgop() System Call Example	9-33
Figure 9-7: Operation Permissions Codes	9-48
Figure 9-8: Control Commands (Flags)	9-48
Figure 9-9: semget() System Call Example	9-53
Figure 9-10: semctl() System Call Example	9-63
Figure 9-11: semop(2) System Call Example	9-74
Figure 9-12: Shared Memory State Information	9-81
Figure 9-13: Operation Permissions Codes	9-85
Figure 9-14: Control Commands (Flags)	9-85
Figure 9-15: shmget(2) System Call Example	9-90
Figure 9-16: shmctl(2) System Call Example	9-97
Figure 9-17: shmop() System Call Example	9-107
Figure 10-1: A Simple curses Program	10-4
Figure 10-2: A Shell Script Using terminfo Routines	10-6
Figure 10-3: The Purposes of initscr() , refresh() , and endwin() in a Program	10-11
Figure 10-4: The Relationship between stdscr and a Terminal Screen	10-15
Figure 10-5: Multiple Windows and Pads Mapped to a Terminal Screen	10-18
Figure 10-6: Input Option Settings for curses Programs	10-45
Figure 10-7: The Relationship Between a Window and a Terminal Screen	10-52
Figure 10-8: Sending a Message to Several Terminals	10-63
Figure 10-9: Typical Framework of a terminfo Program	10-65

Figure 11-1: Object File Format	11-2
Figure 11-2: File Header Contents	11-4
Figure 11-3: File Header Flags (3B2 Computer)	11-5
Figure 11-4: File Header Declaration	11-6
Figure 11-5: Optional Header Contents (3B2, 3B5, 3B15 Computers)	11-7
Figure 11-6: UNIX System Magic Numbers (3B2, 3B5, 3B15 Computers)	11-8
Figure 11-7: aouthdr Declaration	11-9
Figure 11-8: Section Header Contents	11-10
Figure 11-9: Section Header Flags	11-11
Figure 11-10: Section Header Declaration	11-12
Figure 11-11: Relocation Section Contents	11-14
Figure 11-12: Relocation Types (3B2, 3B5, 3B15 Computers)	11-14
Figure 11-13: Relocation Entry Declaration	11-15
Figure 11-14: Line Number Grouping	11-16
Figure 11-15: Line Number Entry Declaration	11-17
Figure 11-16: COFF Symbol Table	11-18
Figure 11-17: Special Symbols in the Symbol Table	11-19
Figure 11-18: Special Symbols (.bb and .eb)	11-20
Figure 11-19: Nested blocks	11-21
Figure 11-20: Example of the Symbol Table	11-22
Figure 11-21: Symbols for Functions	11-22
Figure 11-22: Symbol Table Entry Format	11-23
Figure 11-23: Name Field	11-24
Figure 11-24: Storage Classes	11-25
Figure 11-25: Storage Class by Special Symbols	11-27
Figure 11-26: Restricted Storage Classes	11-27
Figure 11-27: Storage Class and Value	11-28
Figure 11-28: Section Number	11-29
Figure 11-29: Section Number and Storage Class	11-30
Figure 11-30: Fundamental Types	11-32
Figure 11-31: Derived Types	11-33
Figure 11-32: Type Entries by Storage Class	11-34
Figure 11-33: Symbol Table Entry Declaration	11-36
Figure 11-34: Auxiliary Symbol Table Entries	11-37
Figure 11-35: Format for Auxiliary Table Entries for Sections	11-38
Figure 11-36: Tag Names Table Entries	11-39
Figure 11-37: Table Entries for End of Structures	11-39
Figure 11-38: Table Entries for Functions	11-40
Figure 11-39: Table Entries for Arrays	11-41
Figure 11-40: End of Block and Function Entries	11-41
Figure 11-41: Format for Beginning of Block and Function	11-42

List of Figures

Figure 11-42: Entries for Structures, Unions, and Enumerations	11-42
Figure 11-43: Auxiliary Symbol Table Entry	11-44
Figure 11-44: String Table	11-46
Figure 12-1: Operator Symbols	12-5
Figure 12-2: Syntax Diagram for Input Directives	12-32
Figure 13-1: Summary of Default Transformation Path	13-14
Figure 13-2: make Internal Rules	13-26
Figure 14-1: Evolution of an SCCS File	14-7
Figure 14-2: Tree Structure with Branch Deltas	14-8
Figure 14-3: Extended Branching Concept	14-9
Figure 14-4: Determination of New SID	14-20
Figure 15-1: Example of sdb Usage	15-13
Figure 17-1: Escape Sequences for Nongraphic Characters	17-4
Figure 17-2: AT&T 3B Computer Hardware Characteristics	17-7

Purpose

This guide is designed to give you information about programming in a UNIX system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming languages by concentrating on the other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, we are addressing programmers, especially those who have not worked extensively with the UNIX system. No special level of programming involvement is assumed. We hope the book will be useful to people who write only an occasional program as well as those who work on or manage large application development projects.

Programmers in the expert class, or those engaged in developing system software, may find this guide lacks the depth of information they need. For them we recommend the *Programmer's Reference Manual*.

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you might want to look over the *User's Guide* before tackling this one.

Organization

The material is organized into two parts and seventeen chapters, as follows:

- Part 1, Chapter 1 — Overview

Identifies the special features of the UNIX system that make up the programming environment: the concept of building blocks, pipes, special files, shell programming, etc. As a framework for the material that follows, three different levels of programming in a UNIX system are defined: single-user, applications, and systems programming.

- Chapter 2 — Programming Basics

Describes the most fundamental utilities needed to get programs running.

- **Chapter 3 — Application Programming**

Enlarges on many of the topics covered in the previous chapter with particular emphasis on how things change as the project grows bigger. Describes tools for keeping programming projects organized.

- **Part 2, Chapters 4 through 17 — Support Tools, Descriptions, and Tutorials**

Includes detailed information about the use of many of the UNIX system tools.

At the end of the text is an appendix on command packages for the AT&T 3B2 Computer, a glossary, and an index.

The C Connection

The UNIX system supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the UNIX operating system and C has always been and remains very close. Most of the code in the UNIX operating system is C, and over the years many organizations using the UNIX system have come to use C for an increasing portion of their application code. Thus, while this guide is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C.

Hardware/Software Dependencies

The text reflects the way things work on an AT&T 3B2 Computer running UNIX System V at the Release 3.0 level. If you find commands that work a little differently in your UNIX system environment, it may be because you are running under a different release of the software. If some commands just don't seem to exist at all, they may be members of packages not installed on your system. Appendix A describes the command packages available on the AT&T 3B2 Computer. If you do find yourself trying to execute a non-existent command, check Appendix A, then talk to the administrators of your system.

Notation Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout UNIX system documentation:

- Commands that you type in from your terminal are shown in **bold** type.
- Text that is printed on your terminal by the computer is shown in **constant width** type. Constant width type is also used for code samples because it allows the most accurate representation of spacing. Spacing is often a matter of coding style, but is sometimes critical.
- Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display.

Italics are also used to show substitutable values, such as, *filename*, when the format of a command is shown.

- There is an implied RETURN at the end of each command and menu response you enter. Where you may be expected to enter only a RETURN (as in the case where you are accepting a menu default), the symbol **<CR>** is used.
- In cases where you are expected to enter a control character, it is shown as, for example, **CTRL-D**. This means that you press the **d** key on your keyboard while holding down the **CTRL** key.
- The dollar sign, **\$**, and pound sign, **#**, symbols are the standard default prompt signs for an ordinary user and **root** respectively. **\$** means you are logged in as an ordinary user. **#** means you are logged in as **root**.
- When the **#** prompt is used in an example, it means the command illustrated may be used only by **root**.

Command References

When commands are mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses: **command**(section). The numbered sections are located in the following manuals:

- Section (1) *User's Reference Manual*
- Sections (1, 1M), (7), (8) *System Administrator's Reference Manual*
- Sections (1), (2), (3), (4), (5) *Programmer's Reference Manual*

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the UNIX system software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, while we can't say that they have been compiled, we have attempted to maintain the same standards of coding accuracy for them.

Chapter 3: Application Programming

Introduction	3-1
Application Programming	3-2
Numbers	3-2
Portability	3-2
Documentation	3-3
Project Management	3-4
Language Selection	3-5
Influences	3-5
Special Purpose Languages	3-6
What awk Is Like	3-6
How awk Is Used	3-7
Where to Find More Information	3-7
What lex and yacc Are Like	3-7
How lex Is Used	3-8
Where to Find More Information	3-10
How yacc Is Used	3-10
Where to Find More Information	3-12
Advanced Programming Tools	3-13
Memory Management	3-13
File and Record Locking	3-14
How File and Record Locking Works	3-15
lockf	3-17
Where to Find More Information	3-17
Interprocess Communications	3-18
IPC get Calls	3-19
IPC ctl Calls	3-19
IPC op Calls	3-19

Where to Find More Information	3-19
Programming Terminal Screens	3-20
curses	3-20
Where to Find More Information	3-21
 Programming Support Tools	 3-22
Link Edit Command Language	3-22
Where to Find More Information	3-23
Common Object File Format	3-23
Where to Find More Information	3-24
Libraries	3-24
The Object File Library	3-24
Common Object File Interface Macros (ldfcn.h)	3-28
The Math Library	3-29
Shared Libraries	3-32
Where to Find More Information	3-33
Symbolic Debugger	3-33
Where to Find More Information	3-34
lint as a Portability Tool	3-34
Where to Find More Information	3-35
 Project Control Tools	 3-36
make	3-36
Where to Find More Information	3-37
SCCS	3-37
Where to Find More Information	3-39
 liber, A Library System	 3-40

Introduction

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on a UNIX system computer.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially for your own private use (we have called this single-user programming), to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

The next three sections of the chapter deal with a number of loosely-related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 2, Programming Basics, but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

- Advanced Programming deals with such topics as File and Record Locking, Interprocess Communication, and programming terminal screens.
- Support Tools covers the Common Object File Format, link editor directives, shared libraries, SDB, and **lint**.
- Project Control Tools includes some discussion of **make** and SCCS.

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

Application Programming

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a fairly simple application.

When more than one programmer works on a project, there is a need to share such information as:

- the operation of each function
- the number, identity and type of arguments expected by a function
- if pointers are passed to a function, are the objects being pointed to modified by the called function, and what is the lifetime of the pointed-to object
- the data type returned by a function

In an application, there is an odds-on possibility that the same function can be used in many different programs, by many different programmers. The object code needs to be kept in a library accessible to anyone on the project who needs it.

Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, a desirable objective often is to produce code that will run on many different UNIX system computers. Some of the things that affect portability will be touched on later in this chapter.

Documentation

A single-user program has modest needs for documentation. There should be enough to remind the program's creator how to use it, and what the intent was in portions of the code.

On an application development project there is a significant need for two types of internal documentation:

- comments throughout the source code that enable successor programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of 5 or more years, and frequently need to be modified during that time. It is not realistic to expect that the same person who wrote the program will always be available to make modifications. Even if that does happen the comments will make the maintenance job a lot easier.
- hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in the same function being written over again.

Unless end-users have clear, readily-available instructions in how to install and use an application they either will not do it at all (if that is an option), or do it improperly.

The microcomputer software industry has become ever more keenly aware of the importance of good end-user documentation. There are cases on record where the success of a software package has been attributed in large part to the fact that it had exceptionally good documentation. There are also cases where a pretty good piece of software was not widely used due to the inaccessibility of its manuals. There appears to be no truth to the rumor that in one or two cases, end-users have thrown the software away and just read the manual.

Project Management

Without effective project management, an application development project is in trouble. This subject will not be dealt with in this guide, except to mention the following three things that are vital functions of project management:

- tracking dependencies between modules of code
- dealing with change requests in a controlled way
- seeing that milestone dates are met

Language Selection

In this section we talk about some of the considerations that influence the selection of programming languages, and describe two of the special purpose languages that are part of the UNIX system environment.

Influences

In single-user programming the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when making the same decision for an application development project.

Is there an existing standard within the organization that should be observed?

A firm may decide to emphasize one language because a good supply of programmers is available who are familiar with it.

Does one language have better facilities for handling the particular algorithm?

One would like to see all language selection based on such objective criteria, but it is often necessary to balance this against the skills of the organization.

Is there an inherent compatibility between the language and the UNIX operating system?

This is sometimes the impetus behind selecting C for programs destined for a UNIX system machine.

Are there existing tools that can be used?

If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application needs.

Does the application integrate other software into the whole package?

If, for example, a package is to be built around an existing data base management system, there may be constraints on the variety of languages the data base management system can accommodate.

Special Purpose Languages

The UNIX system contains a number of tools that can be included in the category of special purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

What **awk** Is Like

The **awk** utility scans an ASCII input file record by record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the program. The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs consist of only a few. The set of statements may include definitions of the pattern that separates one record from another (a newline character, for example), and what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read, and other actions to be performed after the final record has been read. All statements in between are evaluated in order for each record in the input file. To paraphrase the action of a simple **awk** program, it would go something like this:

Look through the input file.

Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

First do some initialization.

Then, look through the input file.

Every time you see this specific pattern, do this action.

Every time you see this other pattern, do another action.

After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written very quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on a large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

How awk Is Used

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order and pass the resulting rearranged data to a function that adds records to your data base. There is an example of a use of **awk** in the sample application at the end of this chapter.

Where to Find More Information

The manual page for **awk** is in Section (1) of the *User's Reference Manual*. Chapter 4 in Part 2 of this guide contains a description of the **awk** syntax and a number of examples showing ways in which **awk** may be used.

What lex and yacc Are Like

lex and **yacc** are often mentioned in the same breath because they perform complementary parts of what can be viewed as a single task: making sense out of input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be quite similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The UNIX system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing, and that's where **lex** and **yacc** come in. In both utilities, the specifications cause the generation of C language subroutines that deal with streams of characters; **lex** generates subroutines that do lexical analysis while **yacc** generates subroutines that do parsing.

To describe those two tasks in dictionary terms:

Lexical analysis has to do with identifying the words or vocabulary of a language as distinguished from its grammar or structure.

Parsing is the act of describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

Of course, the important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link edited with others to produce programs that then perform whatever process follows from the recognition of the input.

How **lex** Is Used

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a token. The rules are stated in a format that closely resembles the one used by the UNIX system text editor for regular expressions. For example,

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of that rule might have this notation:

```
[ \t]+ ;
```

which, in effect, says to ignore white space. It carries this meaning because no action is specified when a string of one or more blanks or tabs is

recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+ {
    i = atoi(yytext);
    return(NBR);
}
```

This rule depends on several things:

NBR must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is **#include**'d in the declaration section.)

i is declared as an **extern int** in the declaration section.

It is a characteristic of **lex** that things it finds are made available in a character string called **yytext**.

Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is **lex** saying, "Hey, I found the kind of token we call NBR, and its value is now in **i**."

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)
2. **lex.yy.c** is transformed by the C compiler (with a **-c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. **lex.yy.o** is link edited with other subroutines. Presumably one of those subroutines will call **yylex()** with a statement such as:

```
while((token = yylex()) != 0)
```

and other subroutines (or even **main**) will deal with what comes back.

Where to Find More Information

The manual page for **lex** is in Section (1) of the *Programmer's Reference Manual*. A tutorial on **lex** is contained in Chapter 5 in Part 2 of this guide.

How yacc Is Used

yacc subroutines are produced by pretty much the same series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. **y.tab.c** is compiled by the C compiler producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.
3. The object file **y.tab.o** may be link edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by UNIX system editors. **yacc** rules are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined further down the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. An example might make that easier to understand:

```

%token  NUMBER
%%
expr    : numb                { $$ = $1; }
        | expr '+' expr      { $$ = $1 + $3; }
        | expr '-' expr      { $$ = $1 - $3; }
        | expr '*' expr      { $$ = $1 * $3; }
        | expr '/' expr      { $$ = $1 / $3; }
        | '(' expr ')'        { $$ = $2; }
        ;
numb    : NUMBER              { $$ = $1; }
        ;

```

This fragment of a **yacc** specification shows

- **NUMBER** identified as a token in the declaration section
- the start of the rules section indicated by the pair of percent signs
- a number of alternate definitions for *expr* separated by the `|` sign and terminated by the semicolon
- actions to be taken when a rule is matched
- within actions, numbered variables used to represent components of the rule:

`$$` means the value to be returned as the value of the whole rule

`$n` means the value associated with the *n*th component of the rule, counting from the left

- *numb* defined as meaning the token **NUMBER**. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link edited with other subroutines that handle processing that takes place after the parsing—or even ahead of it.

Where to Find More Information

The manual page for `yacc` is in Section (1) of the *Programmer's Reference Manual*. A detailed description of `yacc` may be found in Chapter 6 of this guide.

Advanced Programming Tools

In Chapter 2 we described the use of such basic elements of programming in the UNIX system environment as the standard I/O library, header files, system calls and subroutines. In this section we introduce tools that are more apt to be used by members of an application development team than by a single-user programmer. The section contains material on the following topics:

- memory management
- file and record locking
- interprocess communication
- programming terminal screens

Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that a number of records have been extracted from a data base and need to be held for some further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it is likely to be a great deal more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished; but such an occurrence is not pertinent to this discussion.) There are two C language subroutines available for acquiring blocks of memory and they are both called **malloc**. One of them is **malloc(3C)**, the other is **malloc(3X)**. Each has several related commands that do specialized tasks in the same area. They are:

- **free**—to inform the system that space is being relinquished
- **realloc**—to change the size and possibly move the block
- **calloc**—to allocate space for an array and initialize it to zeros

In addition, **malloc(3X)** has a function, **mallopt**, that provides for control over the space allocation algorithm, and a structure, **mallinfo**, from which the program can get information about the usage of the allocated space.

malloc(3X) runs faster than the other version. It is loaded by specifying

-lmalloc

on the **cc(1)** or **ld(1)** command line to direct the link editor to the proper library. When you use **malloc(3X)** your program should contain the statement

```
#include <malloc.h>
```

where the values for **mallopt** options are defined.

See the *Programmer's Reference Manual* for the formal definitions of the two **mallocs**.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember, in the UNIX system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it, that is, changing any of the data. If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term shared lock is sometimes applied to read locks.

Another distinction needs to be made between mandatory and advisory locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write or create files. This is done through a permission flag established by the file's owner (or the super-user). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed. Thus mandatory may

sound like a simpler and better deal, but it isn't so. The mandatory locking capability is included in the system to comply with an agreement with */usr/group*, an organization that represents the interests of UNIX system users. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term *atomic* is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

How File and Record Locking Works

The system call for file and record locking is **fcntl(2)**. Programs should include the line

```
#include <fcntl.h>
```

to bring in the header file shown in Figure 3-1.

```
/* Flag values accessible to open(2) and fcntl(2) */
/* (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR 2
#define O_NDELAY 04 /* Non-blocking I/O */
#define O_APPEND 010 /* append (writes guaranteed at the end) */
#define O_SYNC 020 /* synchronous write option */

/* Flag values accessible only to open(2) */
#define O_CREAT 00400 /* open with file create (uses third open arg)*/
#define O_TRUNC 01000 /* open with truncation */
#define O_EXCL 02000 /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD 0 /* Duplicate fildes */
#define F_GETFD 1 /* Get fildes flags */
#define F_SETFD 2 /* Set fildes flags */
#define F_GETFL 3 /* Get file flags */
#define F_SETFL 4 /* Set file flags */
#define F_GETLK 5 /* Get file lock */
#define F_SETLK 6 /* Set file lock */
#define F_SETLKW 7 /* Set file lock and wait */
#define F_CHKFL 8 /* Check legality of file flag changes */

/* file segment locking set data type - information passed to system by user */
struct flock {
    short l_type;
    short l_whence;
    long l_start;
    long l_len; /* len = 0 means until end of file */
    short l_sysid;
    short l_pid;
};

/* file segment locking types */
/* Read lock */
#define F_RDLCK 01
/* Write lock */
#define F_WRLCK 02
/* Remove lock(s) */
#define F_UNLCK 03
```

Figure 3-1: The `fcntl.h` Header File

The format of the **fcntl(2)** system call is

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

fildes is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h** includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

lockf

A subroutine, **lockf(3)**, can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

fildes is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek(2)** system call.)

Where to Find More Information

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl(2)**, **fcntl(5)**, **lockf(3)**, and **chmod(2)** in the *Programmer's Reference Manual*. Chapter 7 in Part 2 of this guide is a detailed discussion of the subject with a number of examples.

Interprocess Communications

In Chapter 2 we described **forking** and **execing** as methods of communicating between processes. Business applications running on a UNIX system computer often need more sophisticated methods. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. To go back to the ticket reservation example again for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, "Yes, sir. Just hang on a minute while I start up the reservations program." In transaction driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type the UNIX system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

- | | |
|---------------|---|
| messages | communication is in the form of data stored in a buffer. The buffer can be either sent or received. |
| semaphores | communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25. |
| shared memory | communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there. |

The sets of IPC system calls are:

msgget	semget	shmget
msgctl	semctl	shmctl
msgop	semop	shmop

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET) and removing (IPC_RMID), the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **msgop** has calls that send or receive messages. **semop** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **shmop** has calls that attach or detach shared memory segments.

Where to Find More Information

An example of the use of some IPC features is included in the sample application at the end of this chapter. The system calls are all located in Section (2) of the **Programmer's Reference Manual**. Don't overlook **intro(2)**. It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is contained in Chapter 9 in Part 2 of this guide.

Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the UNIX system. The first of these, **terminfo**, is a data base of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** data base normally begins at the directory **/usr/lib/terminfo**. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the entry for a Teletype 5425 is located in both the file **/usr/lib/terminfo/5/5425** and the file **/usr/lib/terminfo/t/tty5425**.

Describing the capabilities of a terminal can be a painstaking task. Quite a good selection of terminal entries is included in the **terminfo** data base that comes with your 3B2 Computer. However, if you have a type of terminal that is not already described in the data base, the best way to proceed is to find a description of one that comes close to having the same capabilities as yours and building on that one. There is a routine (**setup-term**) in **curses(3X)** that can be used to print out descriptions from the data base. Once you have worked out the code that describes the capabilities of your terminal, the **tic(1M)** command is used to compile the entry and add it to the data base.

curses

After you have made sure that the operating capabilities of your terminal are a part of the **terminfo** data base, you can then proceed to use the routines that make up the **curses(3X)** package to create and manage screens for your application.

The **curses** library includes functions to:

- define portions of your terminal screen as windows
- define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal
- read input from a terminal screen into a program

- write output from a program to your terminal screen
- manipulate the information in a window in a virtual screen area and then send it to your physical screen

Where to Find More Information

In the sample application at the end of this chapter, we show how you might use **curses** routines. Chapter 10 in Part 2 of this guide contains a tutorial on the subject. The manual pages for **curses** are in Section (3X), and those for **terminfo** are in Section (4) of the *Programmer's Reference Manual*.

Programming Support Tools

This section covers UNIX system components that are part of the programming environment, but that have a highly specialized use. We refer to such things as:

- link edit command language
- Common Object File Format
- libraries
- Symbolic Debugger
- **lint** as a portability tool

Link Edit Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. The default locations for the standard Common Object File Format sections are described in **a.out(4)** in the *Programmer's Reference Manual*. On a 3B2 Computer, when an **a.out** file is loaded into memory for execution, the text segment starts at location 0x80800000, and the data section starts at the next segment boundary after the end of the text. The stack begins at 0xC0020000 and grows to higher memory addresses.

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among a lot of other functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Well, the truth is that in the majority of cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two, possibly related, sets of circumstances.

1. Your application is large and consists of a lot of object files.
2. The hardware your application is to run on is tight for space.

Where to Find More Information

Chapter 12 in Part 2 of this guide gives a detailed description of the subject.

Common Object File Format

The details of the Common Object File Format have never been looked on as stimulating reading. In fact, they have been recommended to hardcore insomniacs as preferred bedtime fare. However, if you're going to break into the ranks of really sophisticated UNIX system programmers, you're going to have to get a good grasp of COFF. A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as:

- setting up archive libraries or shared libraries
- using the Symbolic Debugger

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<code><syms.h></code>	symbol table format
<code><linenum.h></code>	line number entries
<code><ldfcn.h></code>	COFF access routines
<code><filehdr.h></code>	file header for a common object file
<code><a.out.h></code>	common assembler and link editor output
<code><scnhdr.h></code>	section header for a common object file
<code><reloc.h></code>	relocation information for a common object file
<code><storclass.h></code>	storage classes for common object files

The object file access routines are described below under the heading "The Object File Library."

Where to Find More Information

Chapter 11 in Part 2 of this guide gives a detailed description of COFF.

Libraries

A library is a collection of related object files and/or declarations that simplify programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common data base can keep the I/O routines in a library that is searched at link edit time.

Prior to Release 3.0 of the UNIX System V the libraries, whether system supplied or application developed, were collections of common object format files stored in an archive (*filename.a*) file that was searched by the link editor to resolve references. Files in the archive that were needed to satisfy unresolved references became a part of the resulting executable.

Beginning with Release 3.0, shared libraries are supported. Shared libraries are similar to archive libraries in that they are collections of object files that are acted upon by the link editor. The difference, however, is that shared libraries perform a static linking between the file in the library and the executable that is the output of **ld**. The result is a saving of space, because all executables that need a file from the library share a single copy. We go into shared libraries later in this section.

In Chapter 2 we described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries, the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, see "The Common Object File Format" in Chapter 11. This library consists of several portions. The functions reside in **/lib/libld.a** and are loaded during the compilation of a

C language program by the `-l` command line option:

```
cc file -lld
```

which causes the link editor to search the object file library. The argument `-lld` must appear after all files that reference functions in `libld.a`.

The following header files must be included in the source code.

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

Function	Reference	Brief Description
ldaclose	ldclose(3X)	Close object file being processed.
ldahread	ldahread(3X)	Read archive header.
ldaopen	ldopen(3X)	Open object file for reading.
ldclose	ldclose(3X)	Close object file being processed.
ldfhread	ldfhread(3X)	Read file header of object file being processed.
ldgetname	ldgetname(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	ldlread(3X)	Prepare object file for reading line number entries via ldlitem .
ldlitem	ldlread(3X)	Read line number entry from object file after ldlinit .
ldlread	ldlread(3X)	Read line number entry from object file.
ldlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	ldlseek(3X)	Seeks to the line number entries of the object file being processed given the name of a section.

Function	Reference	Brief Description
ldnrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	ldshread(3X)	Read section header of the named section of the object file being processed.
ldnsseek	ldsseek(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	ldohseek(3X)	Seeks to the optional file header of the object file being processed.
ldopen	ldopen(3X)	Open object file for reading.
ldrseek	ldrseek(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	ldshread(3X)	Read section header of an object file being processed.
ldsseek	ldsseek(3X)	Seeks to the section of the object file being processed.

Function	Reference	Brief Description
ldtbindex	ldtbindex(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	ldtbread(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	ldtbseek(3X)	Seeks to the symbol table of the object file being processed.
sgetl	sputl(3X)	Access long integer data in a machine independent format.
sputl	sputl(3X)	Translate a long integer into a machine independent format.

Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, which is in the header file **ldfcn.h** (see **ldfcn(4)**). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen(3X)** allocates and initializes the **LDFILE** structure and returns a pointer to the structure. The fields of the **LDFILE** structure may be accessed individually through the following macros:

- The **TYPE** macro returns the magic number of the file, which is used to distinguish between archive files and object files that are not part of an archive.

- The **IOPTR** macro returns the file pointer, which was opened by **ldopen(3X)** and is used by the input/output functions of the C library.
- The **OFFSET** macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.
- The **HEADER** macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an **LDFILE** structure into a reference to its file descriptor field. The available macros are described in **ldfcn(4)** in the *Programmer's Reference Manual*.

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the **-l** option on a command line, as follows:

```
cc file -lm
```

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of each file that uses the routines.

The functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double-precision.

Function	Reference	Brief Description
acos	trig(3M)	Return arc cosine.
asin	trig(3M)	Return arc sine.
atan	trig(3M)	Return arc tangent.
atan2	trig(3M)	Return arc tangent of a ratio.
cos	trig(3M)	Return cosine.
sin	trig(3M)	Return sine.
tan	trig(3M)	Return tangent.

Bessel Functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel(3M)**.

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

Function	Reference	Brief Description
cosh	sinh(3M)	Return hyperbolic cosine.
sinh	sinh(3M)	Return hyperbolic sine.
tanh	sinh(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Function	Reference	Brief Description
ceil	floor (3M)	Returns the smallest integer not less than a given value.
exp	exp (3M)	Returns the exponential function of a given value.
fabs	floor (3M)	Returns the absolute value of a given value.
floor	floor (3M)	Returns the largest integer not greater than a given value.
fmod	floor (3M)	Returns the remainder produced by the division of two given values.
gamma	gamma (3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	hypot (3M)	Return the square root of the sum of the squares of two numbers.

Function	Reference	Brief Description
log	exp(3M)	Returns the natural logarithm of a given value.
log10	exp(3M)	Returns the logarithm base ten of a given value.
matherr	matherr(3M)	Error-handling function.
pow	exp(3M)	Returns the result of a given value raised to another given value.
sqrt	exp(3M)	Returns the square root of a given value.

Shared Libraries

As noted above, beginning with UNIX System V Release 3.0, shared libraries are supported. Not only are some system libraries (**libc** and the networking library) available in both archive and shared library form, but also applications have the option of creating private application shared libraries.

The reason why shared libraries are desirable is that they save space, both on disk and in memory. With an archive library, when the link editor goes to the archive to resolve a reference it takes a copy of the object file that it needs for the resolution and binds it into the **a.out** file. From that point on the copied file is a part of the executable, whether it is in memory to be run or sitting in secondary storage. If you have a lot of executables that use, say, **printf** (which just happens to require much of the standard I/O library) you can be talking about a sizeable amount of space.

With a shared library, the link editor does not copy code into the executable files. When the operating system starts a process that uses a shared library it maps the shared library contents into the address space of the process. Only one copy of the shared code exists, and many processes can use it at the same time.



This fundamental difference between archives and shared libraries has another significant aspect. When code in an archive library is modified, all existing executables are unaffected. They continue using the older version until they are re-link edited. When code in a shared library is modified, all programs that share that code use the new version the next time they are executed.

All this may sound like a really terrific deal, but as with most things in life there are complications. To begin with, in the paragraphs above we didn't give you quite all the facts. For example, each process that uses shared library code gets its own copy of the entire data region of the library. It is actually only the text region that is really shared. So the truth is that shared libraries can add space to executing **a.out**'s even though the chances are good that they will cause more shrinkage than expansion. What this means is that when there is a choice between using a shared library and an archive, you shouldn't use the shared library unless it saves space. If you were using a shared **libc** to access only **strcmp**, for example, you would pick up more in shared library data than you would save by sharing the text.



The answer to this problem, and to others that are somewhat more complex, is to assign the responsibility for shared libraries to a central person or group within the application. The shared library developer should be the one to resolve questions of when to use shared and when to use archive system libraries. If a private library is to be built for your application, one person or organization should be responsible for its development and maintenance.

Where to Find More Information

The sample application at the end of this chapter includes an example of the use of a shared library. Chapter 8 in Part 2 of this guide describes how shared libraries are built and maintained.

Symbolic Debugger



The use of **sdb** was mentioned briefly in Chapter 2. In this section we want to say a few words about **sdb** within the context of an application development project.

sdb works on a process, and enables a programmer to find errors in the code. It is a tool a programmer might use while coding and unit testing a program, to make sure it runs according to its design. **sdb** would normally be used prior to the time the program is turned over, along with the rest of the application, to testers. During this phase of the application development cycle programs are compiled with the `-g` option of **cc** to facilitate the use of the debugger. The symbol table should not be stripped from the object file. Once the programmer is satisfied that the program is error-free, **strip(1)** can be used to reduce the file storage overhead taken by the file.

If the application uses a private shared library, the possibility arises that a program bug may be located in a file that resides in the shared library. Dealing with a problem of this sort calls for coordination by the administrator of the shared library. Any change to an object file that is part of a shared library means the change effects all processes that use that file. One program's bug may be another program's feature.

Where to Find More Information

Chapter 15 in Part 2 of this guide contains information on how to use **sdb**. The manual page is in Section (1) of the *Programmer's Reference Manual*.

lint as a Portability Tool

It is a characteristic of the UNIX system that language compilation systems are somewhat permissive. Generally speaking it is a design objective that a compiler should run fast. Most C compilers, therefore, let some things go unflagged as long as the language syntax is observed statement by statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, there may be real difficulties in running it on some other machine.

That's where **lint** comes in. **lint** produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- cases of disagreement between the type of value expected from a called function and what the function actually returns
- disagreement between the types and number of arguments expected by functions and what the function receives

- inconsistencies that might prove to be bugs
- things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint**.

Code such as this:

```
int i = lseek(fdes, offset, whence)
```

would get by most compilers. However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, it would produce incorrect results, because **i** would contain only the last 16 bits of the value returned.

Since it is reasonable to expect that an application written for a UNIX system machine will be able to run on a variety of computers, it is important that the use of **lint** be a regular part of the application development.

Where to Find More Information

Chapter 16 in Part 2 of this guide contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *Programmer's Reference Manual*.

Project Control Tools

Volumes have been written on the subject of project control. It is an item of top priority for the managers of any application development team. Two UNIX system tools that can play a role in this area are described in this section.

make

make is extremely useful in an application development project for keeping track of what object files need to be recompiled as changes are made to source code files. One of the characteristics of programs in a UNIX system environment is that they are made up of many small pieces, each in its own object file, that are link edited together to form the executable file. Quite a few of the UNIX system tools are devoted to supporting that style of program architecture. For example, archive libraries, shared libraries and even the fact that the **cc** command accepts **.o** files as well as **.c** files, and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**, are all important elements of modular architecture. The two main advantages of this type of programming are that

- A file that performs one function can be re-used in any program that needs it.
- When one function is changed, the whole program does not have to be recompiled.

On the flip side, however, a consequence of the proliferation of object files is an increased difficulty in keeping track of what does need to be recompiled, and what doesn't. **make** is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Once having done that, you conclude a session in which possibly a number of your source code files have been changed by running the **make** command. **make** takes care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

make has the ability to work with files in archive libraries or under control of the Source Code Control System (SCCS).

Where to Find More Information

The **make(1)** manual page is contained in the *Programmer's Reference Manual*. Chapter 13 in Part 2 of this guide gives a complete description of how to use **make**.

SCCS

SCCS is an acronym for Source Code Control System. It consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can:

- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for editing, compiling, etc. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program, **restate**, that was used in several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```
/* Opportunity Cost -- oppty.c */  
  
#include "reodef.h"  
  
float  
oppty(ps)  
struct rec *ps;  
{  
  
    return(ps->i/12 * ps->t * ps->dp);  
}
```

If you decide to add a message to this funtion, you might change the file like this:

```
/* Opportunity Cost -- oppty.c */  
  
#include "reodef.h"  
#include <stdio.h>  
  
float  
oppty(ps)  
struct rec *ps;  
{  
  
    (void) fprintf(stderr, "Opportunity calling\n");  
    return(ps->i/12 * ps->t * ps->dp);  
}
```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change and the login id of the person making the change.

Where to Find More Information

Chapter 14 in Part 2 of this guide is an SCCS user's guide. SCCS commands are in Section (1) of the *Programmer's Reference Manual*.

liber, A Library System

To illustrate the use of UNIX system programming tools in the development of an application, we are going to pretend we are engaged in the development of a computer system for a library. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. It is the way it is only in order to provide some passably realistic examples of UNIX system programming tools in use.

liber is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the data base. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When the system is running the index resides in memory. Semaphores are used to control access to the index. In the pages that follow fragments of some of the system's programs are shown to illustrate the way they work together. The startup program performs the system initialization; opening the semaphores and shared memory; reading the index into the shared memory; and kicking off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

```

/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
    (void) fprintf(stderr, "shmat failed: %d\n", errno);
    exit(1);
}

```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and the book cannot be checked out at two different checkout stations at the same time.

The program fragments do not provide any details on the structure of the index or the book records in the data base.

```

/* liber.h - header file for the
 *          library system.
 */
typedef ... INDEX; /* data structure for book file index */
typedef struct { /* type of records in book file */
    char title[30];
    char author[30];
    .
    .
    .
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;

```

continued

```
/*  startup program */

/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write access to index.
 * 3. Stash id's for shared memory segment and semaphores in a file
 *    where they can be accessed by the programs.
 * 4. Start programs: add-books, card-catalog, and checkout running
 *    on the various terminals throughout the library.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;

main()
{
    .
    .
    .
    if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: shmget failed: errno=%d\n", errno);
        exit(1);
    }
    if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
        exit(1);
    }
}
```

continued

```

if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
{
    (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
    exit(1);
}
(void) fprintf(ipc_file, "%d\n%d\n%d\n", shmid, wrtsem, rdsem);

/*
 * Start the add-books program running on the terminal in the
 * basement. Start the checkout and card-catalog programs
 * running on the various other terminals throughout the library.
 */
.
.
.
}

/* card-catalog program*/

/*
 * 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. Print book record on screen or indicate book was not found.
 * 5. Go to 1.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];

main() {
    .
    .
    .

```

continued

```
while (1)
{
    /*
     * Read author/title/subject information from screen.
     */

    /*
     * Wait for write semaphore to reach 0 (index not being written).
     */
    sop[0].sem_op = 1;
    if (semop(wrtsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n", errno);
        exit(1);
    }
    /*
     * Increment read semaphore so potential writer will wait
     * for us to finish reading the index.
     */
    sop[0].sem_op = 0;
    if (semop(rdsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n", errno);
        exit(1);
    }

    /* Use index to find file pointer(s) for book(s) */

    /* Decrement read semaphore */
    sop[0].sem_op = -1;
    if (semop(rdsem, sop, 1) == -1)
    {
        (void) fprintf(stderr, "semop failed: %d\n", errno);
        exit(1);
    }

    /*
     * Now we use the file pointers found in the index to
     * read the book file. Then we print the information
     * on the book(s) to the screen.
     */
    } /* while */
}
```

continued

```

/*  checkout program */

/*
 * 1. Read screen for Dewey Decimal number of book to be checked out.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise lock
 *    book record and read.
 * 5. If book already checked out print message on screen, otherwise
 *    mark record "checked out" and write back to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */

#include      <stdio.h>
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/sem.h>
#include      <fcntl.h>
#include      "liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;

main()
{
    .
    .
    .
    while (1)
    {
        /*
         * Read Dewey Decimal number from screen.
         */

```

continued

```
/*
 * Wait for write semaphore to reach 0 (index not being written).
 */
sop[0].sem_flg = 0;
sop[0].sem_op = 0;
if (semop(&wrtsem, sop, 1) == -1)
{
    (void) fprintf(stderr, "semop failed: %d\n", errno);
    exit(1);
}
/*
 * Increment read semaphore so potential writer will wait
 * for us to finish reading the index.
 */
sop[0].sem_op = 1;
if (semop(&rdsem, sop, 1) == -1)
{
    (void) fprintf(stderr, "semop failed: %d\n", errno);
    exit(1);
}

/*
 * Now we can use the index to find the book's record position.
 * Assign this value to "bookpos".
 */

/* Decrement read semaphore */
sop[0].sem_op = -1;
if (semop(&rdsem, sop, 1) == -1)
{
    (void) fprintf(stderr, "semop failed: %d\n", errno);
    exit(1);
}

/* Lock the book's record in book file, read the record. */
flk.l_type = F_WRLCK;
flk.l_whence = 0;
flk.l_start = bookpos;
flk.l_len = sizeof(BOOK);
if (fcntl(book_file, F_SETLK, &flk) == -1)
```

continued

```

    {
        (void) fprintf(stderr, "trouble locking: %d\n", errno);
        exit(1);
    }
    if (lseek(book_file, bookpos, 0) == -1)
    {
        Error processing for lseek;
    }
    if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
    {
        Error processing for read;
    }

    /*
     * If the book is checked out inform the client, otherwise
     * mark the book's record as checked out and write it
     * back into the book file.
     */

    /* Unlock the book's record in book file. */
    flk.l_type = F_UNLCK;
    if (fcntl(book_file, F_SETLK, &flk) == -1)
    {
        (void) fprintf(stderr, "trouble unlocking: %d\n", errno);
        exit(1);
    }
} /* while */

/* add-books program*/

/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
 * 3. Use semaphore "wrtsem" to block new readers.
 * 4. Wait for semaphore "rdsem" to reach 0.
 * 5. Insert book into index.
 * 6. Decrement wrtsem.
 * 7. Go to 1.
 */

```

continued

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
    .
    .
    .
    for (;;)
    {

        /*
         * Read information on new book from screen.
         */

        addscr(&bookbuf);

        /* write new record at the end of the bookfile.
         * Code not shown, but
         * addscr() returns a 1 if title information has
         * been entered, 0 if not.
         */

        /*
         * Increment write semaphore, blocking new readers from
         * accessing the index.
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 1;
        if (semop(&bookbuf, sop, 1) == -1)
        {
            (void) fprintf(stderr, "semop failed: %d\n", errno);
            exit(1);
        }
    }
}
```

continued

```

/*
 * Wait for read semaphore to reach 0 (all readers to finish
 * using the index).
 */
sop[0].sem_op = 0;
if (semop(&rdsem, sop, 1) == -1)
{
    (void) fprintf(stderr, "semop failed: %d\n", errno);
    exit(1);
}
/*
 * Now that we have exclusive access to the index we
 * insert our new book with its file pointer.
 */

/* Decrement write semaphore, permitting readers to read index. */
sop[0].sem_op = -1;
if (semop(&wrtsem, sop, 1) == -1)
{
    (void) fprintf(stderr, "semop failed: %d\n", errno);
    exit(1);
}
} /* for */
.
.
.
}

```

The example following, `addscr()`, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
/* addscr is called from add-books.
 * The user is prompted for title
 * information.
 */

#include <curses.h>

WINDOW *cmdwin;

addscr(bb)
struct BOOK *bb;
{
    int c;

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(6, 40, 3, 20);
    mvprintw(0, 0, "This screen is for adding titles to the data base");
    mvprintw(1, 0, "Enter a to add; q to quit: ");
    refresh();
    for (;;)
    {
        refresh();
        c = getch();
        switch (c) {
            case 'a':
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter title: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->title);
                noecho();
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter author: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->author);
                noecho();
```

continued

```

        werase(candwin);
        wrefresh(candwin);
        endwin();
        return(1);
    case 'q':
        erase();
        endwin();
        return(0);
    }
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
        $(CC) $(CFLAGS) -o startup startup.c

add-books: add-books.o addscr.o
        $(CC) $(CFLAGS) -o add-books add-books.o addscr.o

add-books.o: liber.h

checkout: liber.h checkout.c
        $(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
        $(CC) $(CFLAGS) -o card-catalog card-catalog.c

```

Chapter 8: Shared Libraries

Introduction	8-1
Using a Shared Library	8-2
What is a Shared Library?	8-2
The UNIX System Shared Libraries	8-3
Building an a.out File	8-4
Coding an Application	8-4
Deciding Whether to Use a Shared Library	8-5
More About Saving Space	8-6
How Shared Libraries Save Space	8-6
How Shared Libraries Are Implemented	8-9
How Shared Libraries Might Increase Space Usage	8-12
Identifying a.out Files that Use Shared Libraries	8-13
Debugging a.out Files that Use Shared Libraries	8-13
Building a Shared Library	8-14
The Building Process	8-14
Step 1: Choosing Region Addresses	8-14
Step 2: Choosing the Target Library Path Name	8-16
Step 3: Selecting Library Contents	8-17
Step 4: Rewriting Existing Library Code	8-17
Step 5: Writing the Library Specification File	8-17
Step 6: Using mkshlib to Build the Host and Target	8-19
An Example	8-21
Guidelines for Writing Shared Library Code	8-27
Choosing Library Members	8-28
Changing Existing Code for the Shared Library	8-30
Importing Symbols	8-33
Providing Archive Library Compatibility	8-40

Table of Contents

Tuning the Shared Library Code	8-40
Making A Shared Library Upward Compatible	8-43
Summary	8-46

Introduction

With the UNIX system running on smaller machines, such as the AT&T 3B2 Computer, efficient use of disk storage space, memory, and computer power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make **a.out** files (executable object files) smaller on disk storage and processes (**a.out** files that are executing) smaller in memory.

The first part of this chapter, "Using a Shared Library," is designed to help you use UNIX System V shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, "Building a Shared Library," describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers, advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the UNIX system tool **mkshlib(1)** (documented in the *Programmer's Reference Manual*) and how to write C code for shared libraries on a UNIX system. An example is included. Read this part of the chapter only if you have to build a shared library.

NOTE

Shared libraries are a new feature of UNIX System V Release 3.0. An executable object file that needs shared libraries will not run on previous releases of UNIX System V.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains what shared libraries are and how and when to use them on the UNIX system.

What is a Shared Library?

A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. A shared library, like a library that is not shared, is an archive file. For simplicity, however, we refer to an archive file with shared library members as a shared library and one without as an archive library.

When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX system executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

A shared library offers several benefits by not copying code into **a.out** files. It can

- save disk storage space

Because shared library code is not copied into all the **a.out** files that use the code, these files are smaller and use less disk space.

- save memory

By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

"Deciding Whether to Use a Shared Library" in this chapter describes shared libraries in more detail.

The UNIX System Shared Libraries

AT&T provides the C shared library with UNIX System V Release 3.0, and later releases; the networking library included with the Networking Support Utilities is also a shared library. Other shared libraries may be available now from software vendors and in the future from AT&T.

These libraries, like all shared libraries, are made up of two files called the host library and the target library. The host library is the file that the link editor searches when linking programs to create the **.lib** sections in **a.out** files; the target library is the file that the UNIX system uses when running those files. Naturally, the target library must be present for the **a.out** file to run.

Shared Library	Host Library Command Line Option	Target Library Path Name
C Library	<code>-lc_s</code>	<code>/shlib/libc_s</code>
Networking Library	<code>-lnsl_s</code>	<code>/shlib/libnsl_s</code>

Notice the **_s** suffix on the library names; we use it to identify both host and target shared libraries. For example, it distinguishes the standard relocatable C library **libc** from the shared C library **libc_s**. The **_s** also indicates that the libraries are statically linked.

The relocatable C library is still available on the UNIX system; this library is searched by default during the compilation or link editing of C programs. All other archive libraries from previous releases of the system are also available. Just as you use the archive libraries' names, you must use a shared library's name when you want to use it to build your **a.out** files. You tell the link editor its name with the **-l** option, as shown below.

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the UNIX system:

```
cc file.c -o file ... -llibrary_file ...
```

To direct a search of the networking library, for example, you use the following command line.

```
cc file.c -o file ... -lnsl_s ...
```

And to link all the files in your current directory together with the shared C library you'd use the following command line:

```
cc *.c -lc_s
```

Normally, you should include the `-lc_s` argument after all other `-l` arguments on a command line. The shared C library will then be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line are searched.

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

Although there are exceptions, you should avoid redefining standard library routines, such as `printf(3S)` and `strcmp(3C)`. Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

Use only the functions and data mentioned on the manual pages describing the routines in Section 3 of the *Programmer's Reference Manual*. For example, don't try to outsmart the **ctype** design by manipulating the underlying implementation.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program. A well-designed shared library almost always saves space. So, as a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds of library are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section "Coding an Application.") Then compare the two versions of the application for size and performance. For example,

```
$ cat hello.c
main()
{
    printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
unshared: 8680 + 1388 + 2248 = 12316
shared: 300 + 680 + 2248 = 3228
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

When making your decision about using shared libraries, also remember that they are not available on UNIX System V releases prior to Release 3.0. If your program must run on previous releases, you will need to use archive libraries.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the UNIX system
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside and made available outside the library are called exported symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the **a.out** file.

NOTE

Note that the link editor on the UNIX system is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

Although these similarities exist, a shared library differs significantly from an archive library. The major differences relate to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the UNIX system handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor does not copy any code from the library into the program's object file. Instead, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the UNIX system executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 8-1 depicts the **a.out** files produced using a regular archive version and a shared version of the standard C library to compile the following program:

```
main()
{
    ...
    printf( "How do you like this manual?\n" );
    ...
    result = stramp( "I do.", answer );
    ...
}
```

Notice that the shared version is smaller. Figure 8-2 depicts the process images in memory of these two files when they are executed.

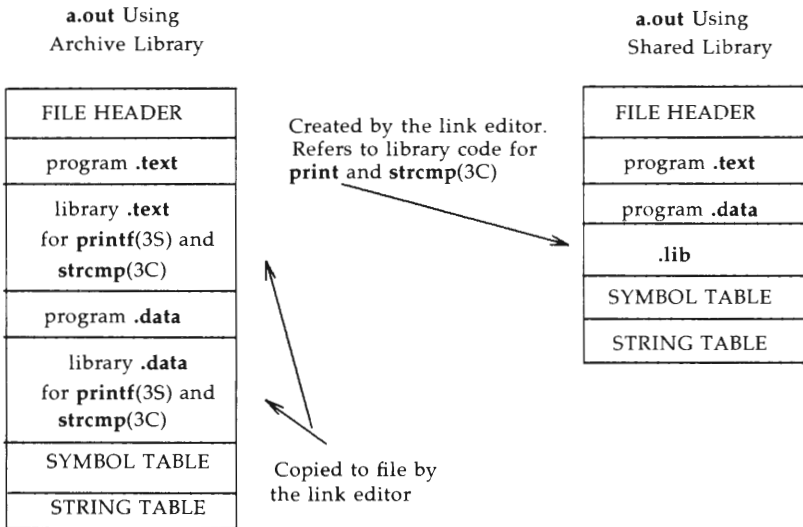


Figure 8-1: **a.out** Files Created Using an Archive Library and a Shared Library

Now consider what happens when several **a.out** files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as indicated in Figure 8-2. This separation enables all processes using the same shared library to reference a single copy of the code.

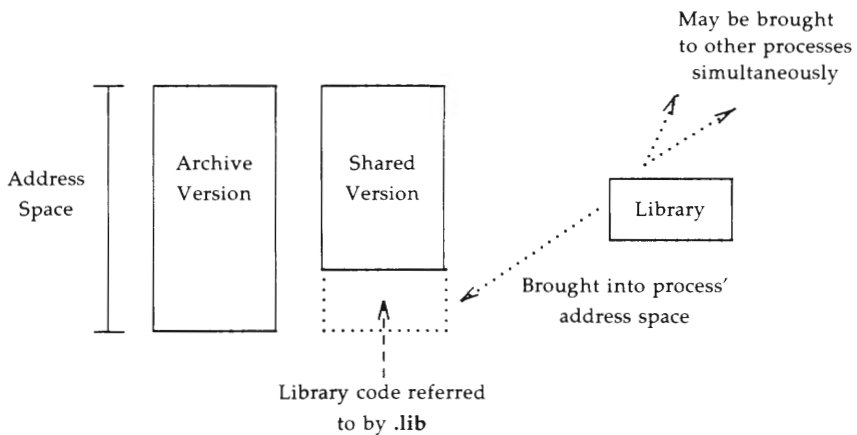


Figure 8-2: Processes Using an Archive and a Shared Library

How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation on the UNIX system to understand how they might increase space usage (this happens seldomly).

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines some text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 8-1 as **.lib**.

The target library used for execution resembles an **a.out** file. The UNIX operating system reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the UNIX system executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the UNIX system uses the same physical code for each process that attaches a shared library's text.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). Processes that share text do not share data and stack area so that they do not interfere with one another.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Also, a process must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To prevent the need to recompile, a shared library is implemented with a branch table on the UNIX system. A branch table associates text symbols with an absolute address that does not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 8-3 shows two **a.out** files executing that make a call to **printf(3S)**. The process on the left was built using an archive library. It already has a copy of the library code defining the **printf(3S)** symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

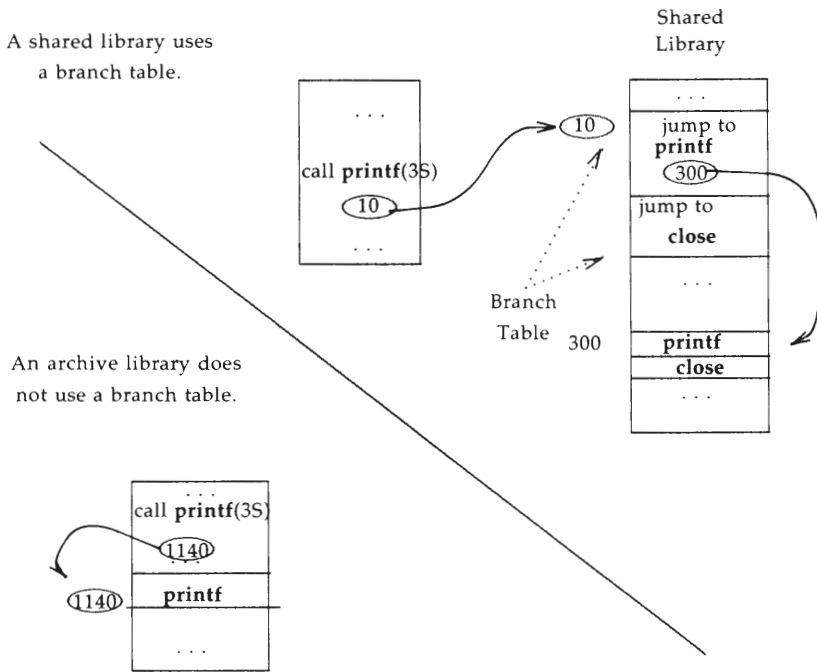


Figure 8-3: A Branch Table in a Shared Library

How Shared Libraries Might Increase Space Usage

A host library might add space to an **a.out** file. Recall that UNIX System V Release 3.0 uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. These symbols might introduce unresolved references during the linking process. To resolve these references, the link editor has to add the **.text** and **.data** sections defining the referenced imported symbols to the **a.out** file. These sections increase the size of the **a.out** file.

A target library might also add space to a process. Again recall from "How Shared Libraries are Implemented" in this chapter that a shared library's target file may have both text and data regions connected to a process. While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process's memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, it would be unwise to use the shared C library to access only **strcmp(3C)**. Although sharing **strcmp(3C)** saves disk storage and memory, the memory cost for sharing all the shared C library's private data region outweighs the savings. The archive version of the library would be more appropriate.

Identifying a.out Files that Use Shared Libraries

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **dump(1)** command (documented in the *Programmer's Reference Manual*) to look at the section headers for the file:

```
dump -hv a.out
```

If the file has a **.lib** section, a shared library is needed. If the **a.out** does not have a **.lib** section, it does not use shared libraries.

With a little more work, you can even tell what libraries a file uses by looking at the **.lib** section contents.

```
dump -L a.out
```

Debugging a.out Files that Use Shared Libraries

Debugging support for shared libraries is currently limited. Shared library data are not dumped to core files, and **sdb(1)** (documented in the *Programmer's Reference Manual*) does not read shared libraries' symbol tables. If you encounter an error that appears not to be in your application code, you may find debugging easier if you rebuild the application with the archive version of the library used.

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps to the building process, the use of the UNIX system tool **mkshlib**(1) which builds the host and target libraries, and some guidelines for writing shared library code.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use the UNIX system shared libraries or other shared libraries that have already been built.

The Building Process

To build a shared library on the UNIX system, you have to complete six major tasks:

- Choose region addresses.
- Choose the path name for the shared library target file.
- Select the library contents.
- Rewrite existing library code to be included in the shared library.
- Write the library specification file.
- Use the **mkshlib** tool to build the host and target libraries.

Here each of these tasks is discussed.

Step 1: Choosing Region Addresses

The first thing you need to do is choose region addresses for your shared library.

Shared library regions on the AT&T 3B2 Computer correspond to memory management unit (MMU) segment size, each of which is 128 KB. The following table gives a list of the segment assignments on the 3B2 Computer (as of the copyright date for this guide) and shows what virtual addresses are available for libraries you might build.

Start Address	Contents	Target Path Name
0x80000000	Reserved for AT&T	
...	UNIX Shared C Library AT&T Networking Library	/shlib/libc_s /shlib/libnsl_s
0x803E0000		
0x80400000	Generic Database Library	Unassigned
0x80420000		
0x80440000	Generic Statistical Library	Unassigned
0x80460000		
0x80480000	Generic User Interface Library	Unassigned
0x804A0000		
0x804C0000	Generic Screen Handling Library	Unassigned
0x804E0000		
0x80500000	Generic Graphics Library	Unassigned
0x80520000		
0x80540000	Generic Networking Library	Unassigned
0x80560000		
0x80580000	Generic – to be defined	Unassigned
...		
0x80660000		
0x80680000	For private use	Unassigned
...		
0x807E0000		

What does this table tell you? First, the AT&T shared C library and the networking library reside at the path names given above and use addresses in the range reserved for AT&T. If you build a shared library that uses reserved addresses you run the risk of conflicting with future AT&T products.

Second, a number of segments are allocated for shared libraries that provide various services such as graphics, database access, and so on. These categories are intended to reduce the chance of address conflicts among commercially available libraries. Although two libraries of the same type may conflict, that doesn't matter. A single process should not usually need to use two shared libraries of the same type. If the need arises, a program can use one shared library and one archive library.

NOTE

Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

Third, several segments are reserved for private use. If you are building a large system with many **a.out** files and processes, shared libraries might improve its performance. As long as you don't intend to release the shared libraries as separate products, you should use the private region addresses. You can put your shared libraries into these segments and avoid conflicting with commercial shared libraries. You should also use the private areas when you will own all the **a.out** files that access your shared library. Don't risk address conflicts.

NOTE

If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the UNIX system without shared library support.

Step 2: Choosing the Target Library Path Name

After you choose the region addresses for your shared library, you should choose the path name for the target library. We chose **/shlib/libc_s** for the shared C library and **/shlib/libnsl_s** for the networking library. (As mentioned earlier, we use the **_s** suffix in the path names of all statically linked shared libraries.) To choose a path name for your shared library, consult the established list of names for your computer or see your system administrator. Also keep in mind that shared libraries needed to boot a UNIX system should normally be located in **/shlib**; other application libraries normally reside in **/usr/lib** or in private application directories. Of course, if your shared library is for personal use, you can choose any convenient path name for the target library.

Step 3: Selecting Library Contents

Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections "Importing Symbols" and "Tuning the Shared Library Code."

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the section "Changing Existing Code for the Shared Library" in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is shown in the next section, "An Example." The contents and format of the specification file are given by the following directives (see also the **mkshlib**(1) manual page):

#address *sectname address*

Specifies the start address, *address*, of section *sectname* for the target file. This directive is typically used to specify the start addresses of the **.text** and **.data** sections.

#target *pathname*

Specifies the path name, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute path name, but it does not have to be.

This directive can be specified only once per shared library specification file.

#branch Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

funcname is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one. The same position cannot be specified more than once, and every position from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as empty slots and can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive can be specified only once per shared library specification file.

#objects Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by white space. This list of objects will be used to build the shared library.

This directive can be specified only once per shared library specification file.

#init *object* Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

```
pimport <white space> import
```

pimport is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each line resembles the C assignment statement:

```
pimport = &import;
```

The assignments set the pointers to default values. All initializations for a particular object file must be given at once and multiple specifications of the same object file are not allowed.

#ident "*string*" Specifies a string, *string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library. Only one **#ident** directive is permitted per shared library specification file.

Specifies a comment. The rest of the line is ignored.

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

Step 6: Using **mkshlib** to Build the Host and Target

The UNIX system command **mkshlib(1)** builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as(1)**, and link editor, **ld(1)**. Tools are invoked through the use of **execvp** (see **exec(2)**) which searches directories in a user's **\$PATH** environment variable. Also, prefixes to **mkshlib** are parsed in much the same manner as prefixes to the **cc(1)** command and invoked tools are given the prefix, where appropriate. For example, **3b2mkshlib** invokes **3b2ld**. These commands all are documented in the *Programmer's Reference Manual*.

The user input to **mkshlib** consists of the library specification file and command line options. We just discussed the specification file; let's take a look at the options now. The shared library build tool has the following syntax:

mkshlib *-s specfil -t target [-h host] [-n] [-q] .DE*

- s *specfil*** Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library, as described in Step 5. Its contents include the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library.
- t *target*** Specifies the name, *target*, of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "Writing the Library Specification File"). If the **-n** option is given, then a new target shared library will not be generated.
- h *host*** Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.
- n** Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **-t** option must still be supplied since a version of the target shared library is needed to build the host shared library.
- q** Suppresses the printing of certain warning messages.

An Example

Follow each of the steps in the library building process to build a small example shared library. While building this library, appropriate guidelines will be displayed amidst text.. Note that the example code is contrived to show samples of problem areas, not to do anything useful.

The name of our library will be **libexam**. Assume the original code was a single source file, as shown below.

```
/* Original exam.c */
#include <stdio.h>

extern int    strlen();
extern char   *malloc(), *strcpy();

int    count    = 0;
char   *Error;

char *
excopy(e)
    char *e;
{
    char *new;

    ++count;
    if ( (new = malloc(strlen(e)+1)) == 0 )
    {
        Error = "no memory";
        return 0;
    }
    return strcpy(new, e);
}

excount()
{
    fprintf(stderr, "excount %d\n", count);
    return count;
}
```

To begin, let's choose the region address spaces for the library's **.text** and **.data** sections from the segments reserved for private use on the 3B2 Computer; note that the region addresses must be on a segment boundary (128K):

```
.text    0x80680000
.data    0x806a0000
```

Also choose the path name for our target library:

/my/directory/libexam_s

Now you need to identify the imported symbols in the library code. (See the guidelines in the section about "Importing Symbols": **malloc**, **strcpy**, **strlen**, **fprintf**, and **_iob**.) A header file defines C preprocessor macros for these symbols; note that you don't use **_iob** directly except through the macro **stderr** from **<stdio.h>**. Also notice the **_libexam_** prefixes for the symbols. The pointers for imported symbols are exported and, therefore, might conflict with other symbols. Using the library name as a prefix reduces the chance of a conflict occurring.

```
/* New file import.h */

#define malloc      (*_libexam_malloc)
#define strcpy      (*_libexam_strcpy)
#define strlen      (*_libexam_strlen)
#define fprintf     (*_libexam_fprintf)
#define _iob        (*_libexam_iob)

extern char         *malloc();
extern char         *strcpy();
extern int          strlen();
extern int          fprintf();
```

NOTE

The file **import.h** does not declare **_iob** as **extern**; it relies on the header file **<stdio.h>** for this information.

You will also need a new source file to hold definitions of the imported symbol pointers. Remember that all global data need to be initialized:

```
/* New file import.c */  
  
#include <stdio.h>  
  
char  (*_libexam_malloc)()   = 0;  
char  (*_libexam_strcpy)()   = 0;  
int    (*_libexam_strlen)()  = 0;  
int    (*_libexam_fprintf)() = 0;  
FILE   (*_libexam_iob)[]     = 0;
```

Next, look at the library's global data to see what needs to be visible externally. (See the guideline "Minimize Global Data.") The variable **count** does not need to be external, because it is accessed through **excount()**. Make it static. (This should have been done for the relocatable version.)

Now the library's global data need to be moved into separate source files. (See the guideline "Define Text and Global Data in Separate Source Files.") The only global datum left is **Error**, and it needs to be initialized. (See the guideline "Initialize Global Data.") **Error** must remain global, because it passes information back to the calling routine:

```
/* New file global.c */  
  
char  *Error   = 0;
```

Integrating these changes into the original source file, we get the following (notice that the symbol names must be declared as **externs**):

```

/* Modified exam.c */
#include "import.h"

#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

static int     count = 0;
extern char     *Error;

char *
excopy(e)
    char *e;
{
    char *new;

    ++count;
    if ( (new = malloc(strlen(e)+1)) == 0 )
    {
        Error = "no memory";
        return 0;
    }
    return strcpy(new, e);
}

excount()
{
    fprintf(stderr, "excount %d\n", count);
    return count;
}

```

NOTE

The new header file **import.h** must be included before **<stdio.h>**.

Next, we must write the shared library specification file for **mkshlib**:

```
/* New file libexam.sl */  
  
1      #target /my/directory/libexam_s  
2      #address .text 0x80680000  
3      #address .data 0x806a0000  
  
4      #branch  
5          excopy          1  
6          exccount        2  
  
7      #objects  
8          import.o  
9          global.o  
10         exam.o  
  
11     #init import.o  
12         _libexam_malloc  malloc  
13         _libexam_strcpy  strcpy  
14         _libexam_strlen  strlen  
15         _libexam_fprintf fprintf  
16         _libexam_iob     _iob
```

Briefly, here is what the specification file does. Line 1 gives the path name of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Lines 2 and 3 give the virtual addresses for the shared library text and data regions, respectively. Line 4 through 6 specify the branch table. Lines 5 and 6 assign the functions **excopy()** and **exccount()** to branch table entries 1 and 2. Only external text symbols, such as C functions, should be placed in the branch table.

Lines 7 through 10 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, a change in the size of static data in **exam.o** would move external data symbols and break compatibility.

Lines 11 through 16 give imported symbol information for the object file **import.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libexam_malloc** will hold a pointer to **malloc**, and so on.

Now, we have to compile the **.o** files as we would for any other library:

```
cc -c import.c global.c exam.c
```

Finally, we need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib -s libexam.sl -t libexam_s -h libexam_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libexam_s.a**, and the target library, **libexam_s**.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility. When appropriate, we describe our experience with building the shared C library to illustrate how following a particular guideline helped us.

We recommend that you read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library. Then use it as a checklist to guide your planning and decision-making.

Before we consider these guidelines, let's consider the restrictions to building a shared library common to all the guidelines. These restrictions involve static linking. Here's a summary of them, some of which are discussed in more detail later. Keep them in mind when reading the guidelines in this section:

- Exported symbols have fixed addresses.

If an exported symbol moves, you have to re-link all **a.out** files that use the library. This restriction applies both to text and data symbols.

- If the library's text changes for one process at run time, it changes for all processes.

Therefore, any library changes that apply only to a single process must occur in data, not in text, because only the data region is private. (Besides, the text region is read-only.)

- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.
- Imported symbols cannot be referenced directly.

Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

Choosing Library Members

Include Large, Frequently Used Routines

These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code. **printf(3S)** and related C library routines (which are documented in the *Programmer's Reference Manual*) are good examples.

When we built the shared C library...

The **printf(3S)** family of routines is used frequently. Consequently, we included **printf(3S)** and related routines in the shared C library.

Exclude Infrequently Used Routines

Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality."

When we built the shared C library...

Our original shared C library had about 44 KB of text. After profiling the code in the library, we removed small routines that were not often used. The current library has under 29 KB of text. The point is that functions used only by a few **a.out** files do not save much disk space by being in a shared library, and their inclusion can cause more paging and decrease performance.

Exclude Routines that Use Much Static Data

These modules increase the size of processes. As "How Shared Libraries are Implemented" and "Deciding Whether to Use a Shared Library" explain, every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

For example, **getgrent(3C)**, which is documented in the *Programmer's Reference Manual*, is not used by many standard UNIX commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library. You can import global data, if necessary, but not local, static data.

Exclude Routines that Complicate Maintenance

All exported symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don't have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of exported data may affect symbol addresses and break compatibility.

Include Routines the Library Itself Needs

It usually pays to make the library self-contained. For example, `printf(3S)` requires much of the standard I/O library. A shared library containing `printf(3S)` should contain the rest of the standard I/O routines, too.

NOTE

This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines mostly explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data from a shared library to an archive library.

Minimize Global Data

In the current shared library implementation, all external data symbols are global; they are visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new exported variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose exported variables were scattered throughout the library modules. Then visible and hidden data would be intermixed. Changing hidden data, such as a string, like **hello** in the following example, moves subsequent data symbols, even the exported symbols:

Before	Broken Successor
<code>int head = 0;</code>	<code>int head = 0;</code>
<code>func()</code>	<code>func()</code>
<code>{</code>	<code>{</code>
<code>...</code>	<code>...</code>
<code>p = "hello";</code>	<code>p = "hello, world";</code>
<code>...</code>	<code>...</code>
<code>}</code>	<code>}</code>
<code>int tail = 0;</code>	<code>int tail = 0;</code>

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus will be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

Adding new exported variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all exported data symbols

and place them at lower addresses than the static (hidden) data. You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
data1.o
...
lastdata.o
text1.o
text2.o
...
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all exported variables' definitions in a single source file without a penalty. You can also use several source files for data definitions.

Initialize Global Data

Initialize exported variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. Using initialized variables is another way to prevent address changes.

The C compilation system on UNIX System V puts uninitialized variables in a common area, and the link editor assigns addresses to them in an unpredictable way. In other words, the order of uninitialized symbols may change from one link editor run to the next. However, the link editor will not change the order of initialized variables, thus allowing a library developer to preserve compatibility.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to re-link all of the **a.out** files that used a previous version of the library.

Importing Symbols

Shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 8-4, the symbols `_libc.ptr1` and `_libc.ptr2` are imported from user's code and the symbol `_libc_malloc` from the library itself.

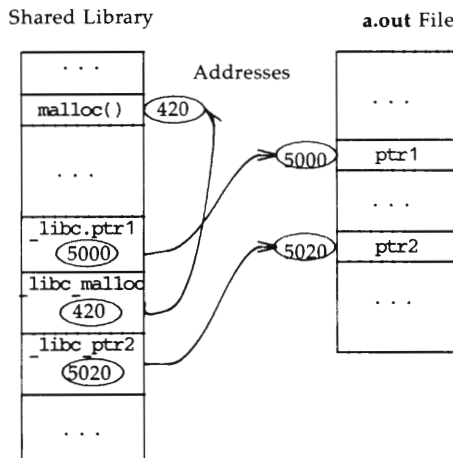


Figure 8-4: Imported Symbols in a Shared Library

The following guidelines describe when and how to use imported symbols.

Imported Symbols that the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. If you leave something out that the library calls, you have to make an imported symbol pointer for it.

Imported Symbols that Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most UNIX commands use the **malloc** from the C library, they can choose either library or define their own.

When we built the shared C library...

Three possible strategies existed for the shared C library. First, we could have excluded **malloc(3X)**. Other library members would have needed it, and so it would have been an imported symbol. This would have worked, but it would have meant less savings.

Second, we could have included the **malloc(3X)** family and not imported it. This would have given us more savings for typical commands, but it had a price. Other library routines call **malloc(3X)** directly, and those calls could not have been overridden. If an application tried to redefine **malloc(3X)**, the library calls would not have used the alternate version. Furthermore, the link editor would have found multiple definitions of **malloc(3X)** while building the application. To resolve this the library developer would have to change source code to remove the custom **malloc(3X)**, or the developer would have to refrain from using the shared library.

Finally, we could have included **malloc(3X)** in the shared library, treating it as an imported symbol. This is what we did. Even though **malloc(3X)** is in the library, nothing else there refers to it directly. If the application does not redefine **malloc(3X)**, both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Let's assume a shared library wants to import the symbol **malloc**. The original archive code and the shared library code appear below.

Archive Code

```
extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Shared Library Code

```
/* See pointers.c on next page */

extern char *(*_libc_malloc)();

export()
{
    ...
    p = (*_libc_malloc)(n);
    ...
}
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **-I** flag to **cpp(1)** would direct the C preprocessor to look in the appropriate directory to find the desired file.

Archive **import.h**

```
/* empty */
```

Shared **import.h**

```
/*
 * Macros for importing
 * symbols. One #define
 * per symbol.
 */

...
#define malloc    (*_libc_malloc)
...
extern char *malloc();
...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in **import.h** actually declares the pointer **_libc_malloc**.

Common Source

```
#include "import.h"

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Alternatively, one can hide the **#include** with **#ifdef**:

Common Source

```
#ifdef SHLIB
#    include "import.h"
#endif

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Of course the transformation is not complete. You must define the pointer **_libc_malloc**.

File **pointers.c**

```
char *(*_libc_malloc)() = 0;
```

Note that `_libc_malloc` is initialized to zero, because it is an exported data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of `malloc` must be assigned to `_libc_malloc`. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the `a.out` file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches `main`. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE

Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into `a.out` files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set `_libc_malloc` to the address of `malloc`:

```
#init pmalloc.o
_libc_malloc    malloc
```

When `mkshlib` builds the host library, it modifies the file `pmalloc.o`, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc;
```

When the link editor extracts `pmalloc.o` from the host library, the relocatable code goes into the `a.out` file. As the link editor builds the final `a.out` file, it resolves the unresolved references and collects all initialization fragments. When the `a.out` file is executed, the run time startup (`crt1`) executes the initialization fragments to set the library pointers.

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will resolve all three, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the symbols.

Let's look at some examples. In the coarse method, a single source file might define all pointers to imported symbols:

Old **pointers.c**

```
int (*_libc_ptr1)() = 0;
char (*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
```

Being able to use them individually requires multiple source files and archive members. Each of the new files defines a single pointer or a small group of related pointers:

File	Contents
ptr1.c	<code>int (*_libc_ptr1)() = 0;</code>
pmalloc.c	<code>char (*_libc_malloc)() = 0;</code>
ptr2.c	<code>int (*_libc_ptr2)() = 0;</code>

Originally, a single object file, **pointers.o**, defines all pointers. Extracting it requires definitions for **ptr1**, **malloc**, and **ptr2**. The modified example lets one extract each pointer individually, thus avoiding the unresolved reference for unnecessary symbols.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. Perhaps the best way to explain this guideline is by example:

When we built the shared C library...

We had an existing archive library to use as the base. This obviously gave us code for individual routines, and the archive library also gave us a model to use for the shared library itself.

We wanted the host library archive file to be compatible with the relocatable archive C library. However, we did not want the shared library target file to include all routines from the archive: including them all would have hurt performance.

Reaching these goals was, perhaps, easier than you might think. We did it by building the host library in two steps. First, we used the available shared library tools to create the host library to match exactly the target. The resulting archive file was not compatible with the archive C library at this point. Second, we added to the host library the set of relocatable objects residing in the archive C library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as UNIX System V Release 3.0. The suggestions come from the experience of building the shared C library.

The archive C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a data base library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Profile the Code

To begin, profile the code that might go into the shared library.

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See "Choosing Library Members" in this chapter for more information.

Organize to Improve Locality

When a function is in **a.out** files, it probably resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow(1)** (documented in the *Programmer's Reference Manual*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline:

When we built the shared C library...

We used a 3B2 Computer to build the library; the architecture of the 3B2 Computer uses 2 KB pages. Using name lists and disassemblies of the shared library target file, we determined where the page boundaries fell.

After grouping related functions, we broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then we used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

After determining the branch table, we rearranged the library's object files without breaking compatibility. We put frequently used, unrelated functions together, because we figured they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on. The following example shows how to change the order of the library's object files:

Before	After
#objects	#objects
...	...
printf.o	stramp.o
fopen.o	malloc.o
malloc.o	printf.o
stramp.o	fopen.o
...	...

Avoid Hardware Thrashing

Finally, you may have to consider the hardware you're using to obtain better performance. Using the 3B2 Computer, for example, you need to consider its memory management. Part of the memory management hardware is an 8-entry cache for translating virtual to physical addresses. Each segment (128 KB) is mapped to one of the eight entries. Consequently, segments 0, 8, 16, ... use entry 0; segments 1, 9, 17, ... use entry 1; and so on.

You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry, again.

When we built the shared C library...

We avoided the cache entry conflicts. At least with the 3B2 Computer architecture, a library's text and data segment numbers should differ by something other than eight.

Making A Shared Library Upward Compatible

The following guidelines explain how to build upward-compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, versions of the library may be incompatible with each other. This may complicate development, but it is possible.

Comparing Previous Versions of the Library

Shared library developers normally want newer versions of a library to be compatible with previous ones. As mentioned before, **a.out** files will not execute properly otherwise.

The following procedures let you check libraries for compatibility. In these tests, two libraries are said to be compatible if their exported symbols have the same addresses. Although this criterion usually works, it is not foolproof. For example, if a library developer changes the number of arguments a function requires, the new function may not be compatible with the old. This kind of change may not alter symbol addresses, but it will break old **a.out** files.

Let's assume we want to compare two target shared libraries: **new.libx_s** and **old.libx_s**. We use the **nm(1)** command to look at their symbols and **sed(1)** to delete everything except external symbols. A small **sed** program simplifies the job.

New file **cmplib.sed**

```
sed      '/!extern!.*!d
s///
/^.bt/d
/^etext /d
/^edata /d
/^end /d'
```

The first line of the **sed** script deletes all lines except those for external symbols. The second line leaves only symbol names and values in the output. The last four lines delete special symbols that have no bearing on library compatibility; they are not visible to application programs. You will have to create your own file to hold the **sed** script.

Now we are ready to create lists of symbol names and values for the new and old libraries:

```
nm old.libx_s | sed -f cmplib.sed >old.nm
nm new.libx_s | sed -f cmplib.sed >new.nm
```

Next, we compare the symbol values to identify differences:

```
diff old.nm new.nm
```

If all symbols in the two libraries have the same values, the **diff(1)** command will produce no output, and the libraries are compatible. Otherwise, some symbols are different and the two libraries may be incompatible. **diff(1)**, **nm(1)**, and **sed(1)** are documented in the *User's Reference Manual*.

Dealing with Incompatible Libraries

When you determine that two libraries are incompatible, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target path name to the new version of the library. The host and target path names are independent; so you don't have to change the host library path name. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library path name for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

NOTE

You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

Summary

This chapter described the UNIX system shared libraries and explained how to use them. It also explained how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the UNIX system on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.



Glossary

Ada	Named after the Countess of Lovelace, the nineteenth century mathematician and computer pioneer, Ada is a high-level general-purpose programming language developed under the sponsorship of the U.S. Department of Defense. Ada was developed to provide consistency among programs originating in different branches of the military. Ada features include packages that make data objects visible only to the modules that need them, task objects that facilitate parallel processing, and an exception handling mechanism that encourages well-structured error processing.
ANSI standard	ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.
a.out file	a.out is the default file name used by the link editor when it outputs a successfully compiled, executable file. a.out contains object files that are combined to create a complete working program. Object file format is described in Chapter 11, "The Common Object File Format," and in a.out(4) in the <i>Programmer's Reference Manual</i> .
application program	An application program is a working program in a system. Such programs are usually unique to one type of users' work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.
archive	An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command ar(1) collects data for use as a library.

argument An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments: options and operands. Options are immediately preceded by a minus sign (–) and change the execution or output of the command. Some options can themselves take arguments. Operands are preceded by a space and specify files or directories that will be operated on by the command. For example, in the command

pr –t –h Heading file

all of the elements after the **pr** are arguments. **–t** and **–h** are options, **Heading** is an argument to the **–h** option, and **file** is an operand.

For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if more than two are present they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a page in Section 3 of the *Programmer's Reference Manual*, describes the number and data type of argument(s) expected by the function.

ASCII ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX system. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.

assembler The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many

cases, the assembly language instructions map 1 to 1 with the binary machine instructions.



assembly language

A programming language that uses the instruction set that applies to a particular computer.

BASIC

BASIC is a high-level conversational programming language that allows a computer to be used much like a complex electronic calculating machine. The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.

branch table

A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.



buffer

A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents are read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.

byte

A byte is a unit of storage in the computer. On many UNIX systems, a byte is eight bits (binary digits), the equivalent of one character of text.

byte order

Byte order refers to the order in which data are stored in computer memory.



C

The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both

high-level and low-level tasks. Although it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-processing, and data base programs. The C programming language was designed for and implemented on the UNIX system; however, the language is not limited to any one operating system or machine.

C compiler	The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.
C preprocessor	The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (#) are directives to the preprocessor. Command line options of the <code>cc(1)</code> command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.
CCS	CCS is an acronym for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, then passes the processed code to the C compiler, which produces assembly language code that it passes to the assembler. The assembler in turn produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an optimizer that makes the code produced as efficient as possible, productivity tools, tools used to read and manipulate object files, and libraries that provide runtime support, access to system calls, input/output, string manipulation, mathematical functions, and other code processing functions.

- COBOL** COBOL is an acronym for COMmon Business Oriented Language. COBOL is a high-level programming language designed for business and commercial applications. The English-language statements of COBOL provide a relatively machine-independent method of expressing a business-oriented problem to the computer.
- COFF** COFF is an acronym for Common Object File Format. COFF refers to the format of the output file produced on some UNIX systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features:
- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
 - Space is provided for symbolic information used by debuggers and other applications.
 - Users may make some modifications in the object file construction at compile time.
- command** A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.
- command line** A command line is composed of the command name followed by any argument(s) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis in a notation designed to show the correct way to type in a command, with or without options and arguments.

compiler	A compiler transforms the high-level language instructions in a program (the source code) into object code or assembly language. Assembly language code may then be passed to the assembler for further translation into machine instructions.	
core	Core is a (mostly archaic) synonym for primary memory.	
core file	A core file is an image of a terminated process saved for debugging. A core file is created under the name "core" in the current directory of the process when an abnormal event occurs resulting in the process' termination. A list of these events is found in the signal(2) manual page in section 2 of the <i>Programmer's Reference Manual</i> .	
core image	Core image is a copy of all the segments of a running or terminated program. The copy may exist in main storage, in the swap area, or in a core file.	
curses	curses(3X) is a library of C routines that are designed to handle input, output, and other operations in screen management programs. The name curses comes from the cursor optimization that the routines provide. When a screen management program is run, cursor optimization minimizes the amount of time a cursor has to move about a screen to update its contents. The program refers to the terminfo(4) data base at run time to obtain the information that it needs about the screen (terminal) being used. See terminfo(4) in the <i>Programmer's Reference Manual</i> .	
data symbol	A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.	
data base	A data base is a bank of information on a particular subject or subjects. On-line data bases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.	

-  **debug** Debugging is the process of locating and correcting errors in computer programs.
-  **default** A default is the way a computer will perform a task in the absence of other instructions.
- delimiter** A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX system command syntax, a minus sign (-) is most often the delimiter for option names, for example, `-s` or `-n`, although some commands also use a plus sign (+).
-  **directory** A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization.
- The `ls(1)` command is used to list the contents of a directory. When you first log onto the system, you are in your home directory (`$HOME`). You can move to another directory by using the `cd(1)` command and you can print the name of the current directory by using the `pwd(1)` command. You can also create new directories with the `mkdir(1)` command and remove empty directories with `rmdir(1)`.
- A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative path name or the full path name of a directory.
-  **dynamic linking** Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.

environment	An environment is a collection of resources used to support a function. In the UNIX system, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer. An environment variable is a shell variable such as <code>\$HOME</code> (which stands for your login directory) or <code>\$PATH</code> (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to work. These variables come from <code>/etc/profile</code>, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal <code>.profile</code> file, which you create in your login directory to tailor your own working environment. You can also temporarily set variables at the shell level.	 
executable file	An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the <code>chmod(1)</code> command. In addition to being ready to run, a file in the UNIX system needs to have execute permission.	
exit	A specific system call that causes the termination of a process. The <code>exit(2)</code> call will close any open files and clean up most other information and memory which was used by the process.	

exit status: return code

An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.

exported symbol

A symbol that a shared library defines and makes available outside the library. See imported symbol.

expression

An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.

file

A file is an identifiable collection of information that, in the UNIX system, is a member of a file system. A file is known to the UNIX system as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special or directory files.

file and record locking

The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.

file descriptor

A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; file descriptor 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).

file system	A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.
filter	A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are grep(1) and tail(1) , which select and output part of the input; sort(1) , which sorts the input; and wc(1) , which counts the number of words, characters, and lines in the input. sed(1) and awk(1) are also filters but they are called programmable filters or data transformers because a program must be supplied as input in addition to the data to be transformed.
flag	A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX system flags are usually indicated by a leading hyphen (-). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean to point out or to draw attention to. See option.
fork	fork(2) is a system call that divides a new process into two, the parent and child processes, with separate, but initially identical, text, data, and stack segments. After the duplication, the child (created) process is given a return code of 0 and the parent is given the process id of the newly created child as the return code.

FORTTRAN	FORTTRAN is an acronym for FORMula TRANslator. FORTRAN is a high-level programming language originally designed for scientific and engineering calculations but now also widely adapted for many business uses.
function	A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.
header file	A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.
high-level language	A high-level language is a computer programming language such as C, FORTRAN, COBOL, or PASCAL that uses symbols and command statements representing actions the computer is to perform, the exact steps for a machine to follow. A high-level language must be translated into machine language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of machine language instructions. The low-level details of the computer's internal organization are left to the compilation system.
host machine	A host machine is the machine on which an a.out file is built.

imported symbol	A symbol used but not defined by a shared library. See exported symbol.
interpreted language	An interpreted language is a high-level language that is not translated by a compilation system and stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.
Interprocess Communication	Interprocess Communication describes software that enables independent processes running at the same time, to exchange information through messages, semaphores, or shared memory.
interrupt	An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX programs by pressing the delete or break keys, by typing Control-d, or by using the kill(1) command.
I/O (Input/Output)	I/O is the process by which information enters (input) and leaves (output) the computer system.
kernel	The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX system relies. It is responsible for most operating system functions. It schedules and manages the work done by the computer and maintains the file system. The kernel has its own text, data, and stack areas.
lexical analysis	Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as = , := , and ; . Lexical

analysis enables you to recognize, for example, that the stream of characters `'print("hello, universe")'` is to be analyzed into a series of tokens beginning with the word `'print'` and not with, say, the string `'print("h.'` In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).

library

A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.

link editor

A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX system, but you can also link edit previously compiled files by using the **ld(1)** command.

magic number

The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether shared or non-shared text, and on which processor the file is executable.

makefile

A makefile is a file that lists dependencies among the source code files of a software product and methods for updating them, usually by recompilation. The **make(1)** command uses the makefile to maintain self-consistent software.

manual page

A manual page, or "man page" in UNIX system jargon, is the repository for the detailed description of a command, a system call, subroutine or other UNIX system component.

null pointer	A null pointer is a C pointer with a value of 0.
object code	Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file that is ready to run is an executable file.
optimizer	An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by and speeds the execution time of the code.
option	An option is an argument used in a command line to modify program output by modifying the execution of a command. An option is usually one character preceded by a hyphen (-). When you do not specify any options, the command will execute according to its default options. For example, in the command line

ls -a -l directory

	-a and -l are the options that modify the ls(1) command to list all directory entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).
parent process	A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.
parse	To parse is to analyze a sentence in order identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are analyzed.
PASCAL	PASCAL is a multipurpose high-level programming language often used to teach programming. It is based on the ALGOL programming language and emphasizes structured programming.

 path name

A path name is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the path name begins with a slash, it is called a full path name; the initial slash means that the path begins at the **root** directory.

A path name that does not begin with a slash is known as a relative path name, meaning relative to the present working directory. A relative path name may begin either with a directory name or with two dots followed by a slash (../). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with ../ indicates that the path first proceeds up the hierarchy; ../ is the parent of the present working directory.

 permissions

Permissions are a means of defining a right to access a file or directory in the UNIX file system. Permissions are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions:

- ☐ Read permission (r) includes permission to cat, pg, lp, and cp a file.
- ☐ Write permission (w) is the permission to change a file.
- ☐ Execute permission (x) is the permission to run an executable file.

Permissions can be changed with the UNIX system **chmod(1)** command.

 pipe

A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe

symbol (`|`), which indicates that the output from the process on the left should be routed to the process on the right. For example, in the command

who | wc -l,

the output from the **who**(1) command, which lists the users who are logged on to the system, is used as input for the word-count command, **wc**(1), with the **l** option. The result of this pipeline (succession of commands connected by pipes) is the number of people who are currently logged on to the system.

portable

Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other systems, time is wasted, and each modification increases the chances for an error.

preprocessor

Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn**(1) and **tbl**(1) are preprocessors for **nroff**(1). **grap**(1) is a preprocessor for **pic**(1). **c++**(1) is a preprocessor for the C compiler.

process

A process is a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines.

The process id is a unique system-wide identification number that identifies an active process. The process status command, **ps**(1), prints the process ids of the processes that belong to you.

program	A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting on the status of the system. A subprogram is part of a larger program and can be compiled independently.
regular expression	A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to be searched for in a file. The pattern-matching functions of ed(1) and grep(1) , for example, use regular expressions.
routine	A routine is a discrete section of a program to accomplish a set of related tasks
semaphore	In the UNIX system, a semaphore is a sharable short unsigned integer maintained through a family of system calls which include calls for increasing the value of the semaphore, setting its value, and for blocking waiting for its value to reach some value. Semaphores are part of the UNIX system IPC facility.
shared library	Shared libraries include object modules that may be shared among several processes at execution time.
shared memory	Shared memory is an IPC (interprocess communication) facility in which two or more processes can share the same data space.
shell	The shell is the UNIX system program— sh(1) —responsible for handling all interaction between you and the system. It is a command language interpreter that understands your commands and causes the computer to act on them. The shell also establishes the environment at your terminal. A shell normally is started for you as part of the login process. Three shells, the Bourne shell, the Korn shell and the C shell, are popular. The shell can also be used as a programming language to write procedures for a variety of tasks.

signal: signal number

A signal is a message that you send to processes or processes send to one another. The most common signals you might send to a process are ones that would cause the process to stop: for example, interrupt, quit, or kill. A signal sent by a running process is usually a sign of an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.

source code

Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.

standard error

Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX system, the default case is to associate standard error with the user's terminal.

standard input

Standard input is an input stream to a program. In the UNIX system, the default case is to associate standard input with the user's terminal.

standard output

Standard output is an output stream from a program. In the UNIX system, the default case is to associate standard output with the user's terminal.

stdio: standard input-output

stdio(3S) is a collection of functions for formatted and character-by-character input-output at a higher level than the basic read, write, and open operations.

static linking

Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.

stream

- A stream is an open file with buffering provided by the stdio package.

- A stream is a full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character input/output interface for the user processes.

string

A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.

strip

strip(1) is a command that removes the symbol table and relocation bits from an executable file.

subroutine

A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.

symbol table

A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.

symbol value

The value of a symbol, typically its virtual address, used to resolve references.

syntax

- Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command on the command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command.

- Language syntax is the set of rules that describe how the elements of a programming language may legally be used.

system call

A system call is a request by an active process for a service performed by the UNIX system kernel, such as I/O, process creation, etc. All system operations are allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid(2)** system call allows you to inspect information that is not normally available since it resides in the operating system's address space.

target machine

A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.

TCP/IP (Transmission Control Protocol/Internetwork Protocol)

TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.

terminal definition

A terminal definition is an entry in the **terminfo(4)** data base that describes the characteristics of a terminal. See **terminfo(4)** and **curses(3X)** in the *Programmer's Reference Manual*.

terminfo

- a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys.
- a data base containing the compiled descriptions of many terminals that can be used with **curses(3X)** screen management programs. These

descriptions specify the capabilities of a terminal and how it performs various operations — for example, how many lines and columns it has and how its control characters are interpreted. A **curses(3X)** program refers to the data base at run time to obtain the information that it needs about the terminal being used.

See **curses(3X)** in the *Programmer's Reference Manual*. **terminfo(4)** routines can be used in shell programs, as well as C programs.

text symbol

A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.

tool

A tool is a program, or package of programs, that performs a given task.

trap

A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX operating system

The UNIX operating system is a general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T. An operating system is the software on the computer under which all other software runs. The UNIX operating system has two basic parts:

- The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.
- The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

	The utility programs or UNIX system commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.	
userid	A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.	
utility	A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software errors, for example, a loader, editor, debugging, or diagnostics package.	
variable	❑ A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next. ❑ A variable in the shell is a name representing a string of characters (a string value). ❑ A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter). ❑ A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.	
white space	White space is one or more spaces, tabs, or newline characters. White space is normally used to separate strings of characters, and is required to separate the command from its arguments on a command line.	
window	A window is a screen within your terminal screen that is set off from the rest of the screen. If you have two windows on your screen, they are independent of each other and the rest of the screen.	

The most common way to create windows on a UNIX system is by using the layers capability of the TELETYPE 5620 Dot-Mapped Display. Each window you create with this program has a separate shell running it. Each one of these shells is called a layer.

If you do not have this facility, the **shl(1)** command, which stands for shell layer, offers a function similar to the layers program. You cannot create windows using **shl(1)**, but you can start different shells that are independent of each other. Each of the shells you create with **shl(1)** is called a layer.

word

A word is a unit of storage in a computer that is composed of bytes of information. The number of bytes in a word depends on the computer you are using. The AT&T 3B Computers, for example, have 32 bits or 4 bytes per word, and 16 bits or 2 bytes per half word.

Index

- action ... 4:1, 6-8, 17, 20, 22, 25, 29-30, 34-41, 44; 5:1, 3-8, 10, 12, 14-15, 17; 6:1, 6-10, 13-15, 17, 19, 22-23, 25, 27-31, 33-34, 36, 38-42, 49
- advisory ... 7:1-3, 18-19
- archive ... 2:12-13, 76-77, 82; 3:23-24, 28-29, 32-33, 36-37; 8:2-9, 11, 13-16, 27, 30-31, 34-41
- argument ... 2:15-18, 72, 76; 3:25; 13:23; 14:11, 26, 32, 35, 37; 15:2, 7; 16:2, 8; 17:14, 35, 55
- array ... 15:5; 16:8; 17:5, 8, 13-14, 16-19, 27-28, 30-31, 34-36, 45, 54-55, 57
- assembler ... 1:2; 2:9
- assembly language ... 2:5, 7, 9, 11, 55
- assignment ... 13:4, 7, 23; 14:2, 13, 27, 39; 16:5, 8-9, 11-12; 17:8-9, 16, 19, 21-22, 32, 34, 42, 52, 58, 60
- awk ... 2:6-7; 3:6-7; 4:1-2, 4, 8, 10-11, 15, 17-25, 27-32, 34-40, 42, 44-50
- basic ... 3:13
- bc ... 2:7
- bessel ... 3:30
- braces ... 17:5, 33-35
- break ... 13:16; 14:39; 16:2, 5-6; 17:2, 41, 64
- bss ... 2:9, 72
- cast ... 16:8-9; 17:11, 14, 16, 35, 50, 52, 57
- cc ... 2:10-13, 58-59, 70; 3:14, 25, 29, 34, 36
- cflow ... 2:55
- COFF ... 3:23-24; 11:1, 3, 6; 12:13
- context ... 2:18; 3:33
- ctrace ... 2:58-59, 62
- curses ... 2:8; 3:20-21, 49; 10:1-7, 9-14, 17-44, 46-51, 55-66, 69-70, 73, 83-84, 91, 95, 97, 100
- curses/terminfo ... 10:1, 3, 7
- cursor ... 10:1, 3-4, 12-13, 19, 25-26, 28, 44, 64, 73, 75, 77, 82-83, 95
- cxref ... 2:62
- dc ... 2:7
- deadlock ... 7:17, 20
- debugging ... 13:14; 15:1-3, 6, 8, 11-12
- delta ... 14:1-9, 12-13, 15-19, 22-28, 30-37, 40-41
- description ... 13:1-4, 6-9, 12-15, 22-23; 14:10, 12, 24-25, 29, 37-38; 16:3; 17:49
- editor ... 1:3; 2:74, 76; 3:1, 8-9, 14, 22-25, 29, 32
- environment ... 1:1-8; 2:1, 5-6, 9, 18-19; 3:1-2, 5, 13, 22, 36
- error ... 2:12, 17, 28, 62, 69; 3:14
- etc ... 1:3; 3:37
- exclusive ... 7:2, 9
- exec ... 2:18, 41-43; 3:18
- expression ... 13:10, 17; 15:7; 16:8, 11-12; 17:5, 8-23, 26-29, 31, 33-34, 37-42, 50, 52-55, 57, 59, 63-64
- extern ... 17:24-25, 36-38, 44-45, 47, 61, 65
- fcntl ... 3:15, 17; 7:1, 6, 9, 11, 13-15, 17, 20
- fgets ... 2:29-30
- fildev ... 3:17

- float ... 15:4-5, 12; 16:8; 17:2, 4, 7-10, 13-14, 25, 28, 34-35, 45, 57-58, 61
- fork ... 2:41, 42-44; 3:18
- function ... 15:2-11; 16:2-8, 11-12; 17:6, 8, 11-14, 19-20, 23-25, 27-28, 33, 36-38, 42-47, 52-53, 55-58
- get ... 13:20; 14:3-6, 12-19, 22-24, 26, 29-30, 32, 34-37, 40
- header ... 2:9, 33, 50, 72, 75; 3:9, 13, 15, 18, 23-25, 28-29, 49
- host ... 5:1, 6; 8:3, 6, 9-10, 12, 14, 17, 19-20, 27, 34, 38, 45
- import ... 8:6, 12, 23-24, 26-30, 32-39
- int ... 16:8-9, 11; 17:2-4, 7-8, 10-11, 13-15, 18-21, 25, 27-28, 30, 34, 36-37, 40, 44-45, 47, 55, 58, 61
- Inter-Process ... 9:1
- IPC ... 3:18-19
- ld ... 2:12; 3:14, 22, 24, 36
- lex ... 2:7, 55; 3:6-11; 4:4; 5:1-4, 6-10, 12-15, 17-20; 6:12
- lexical ... 4:2, 8-9; 5:1-3, 5, 8, 12, 15, 17-19; 6:1-4, 6-7, 9-12, 15, 32, 36-37, 40, 42, 45, 50
- libc ... 2:12, 19, 76; 3:24, 32-33
- library ... 4:25, 43; 5:15, 18-19; 6:32-33, 50; 7:1, 6; 8:1-20, 23-24, 26-38, 40-41, 43-46; 13:2, 8, 10-11, 15-17; 16:1-3; 17:12
- Link ... 11:13; 12:1, 4, 6, 15, 26
- link editor ... 2:9, 11-14, 36
- lint ... 2:69; 3:1, 22, 34-35; 16:1-12
- lockf ... 3:17; 7:1-20; 9:81, 83, 94, 96
- locking ... 1:8; 3:13-15
- lvalue ... 17:8, 13-16, 21-22, 54, 59
- macro ... 2:7, 74-75; 3:28-29
- make ... 2:74-76; 3:1, 36-37; 13:1-7, 9-20, 22-24
- makefile ... 13:1-2, 9-10, 12, 15-16, 19-21, 23-24
- malloc ... 3:13-14
- mandatory ... 5:3; 7:1, 3-4, 18-19
- mathematical ... 2:4, 7
- message ... 9:1-40, 44, 51, 60-62, 73, 80, 88-89, 96, 106
- mkshlib ... 8:1, 14, 17, 26-27, 38
- object ... 2:9-10, 12, 14, 18-19, 34, 72; 3:2, 8-11, 22-25, 28-29, 32, 34, 36
- options ... 2:10-13, 50, 58, 69, 75; 3:14
- parent ... 2:42-43
- parser ... 5:12, 15-19; 6:1, 3, 6-7, 9-15, 17-24, 27-29, 31-35, 37-38, 40, 49-50
- parsing ... 3:5, 8, 11
- Pascal ... 2:4
- pattern ... 4:1-2, 4, 6, 8, 17, 22, 25, 29-34, 44, 47; 5:1, 3, 6-9, 11, 14
- pipes ... 1:2-3
- portability ... 2:3, 69; 3:2, 22, 35
- preprocessor ... 2:7
- prof ... 2:10, 70-71
- register ... 15:11-12; 16:4-5; 17:2, 6, 24-25, 33-34, 38, 41, 44-45, 58, 61
- relational ... 16:8; 17:18-19
- return ... 13:9, 22; 14:37; 15:10-11; 16:3, 5-9; 17:2, 8, 13-14, 20-21, 27-28, 36-37, 42, 44, 52, 56, 64
- SCCS ... 2:83; 3:1, 37-39; 13:13, 18-20; 14:1-3, 5-7, 9-19, 22-30, 32-41
- screen ... 10:1, 3-4, 6, 9-14, 17, 19, 25-29, 31, 33, 36, 38, 40, 43-44, 50-51, 55-62, 64-65, 73-75, 77, 79-80, 83-84, 91, 93, 95, 100

- sdb ... 2:9-10, 12, 72-73; 3:33-34;
15:1-12
- section ... 10:1-2, 4, 9-11, 13, 18, 25,
33, 36, 38-39, 49, 56, 66, 68, 70,
72, 75; 11:1-3, 9-10, 12-15, 29-
30, 38, 47; 12:1-4, 6-26, 28-30
- semaphore ... 9:1, 40-47, 49-52, 55-
62, 70-73
- shared ... 1:8; 3:1, 14, 18-19, 23-24,
32-34, 36, 40; 4:37; 5:5; 7:2, 16-
17; 8:1-17, 20, 26-38, 40-41, 43-
46; 9:1, 78-84, 86-89, 92-96,
103-106
- shell ... 1:4-5, 9; 2:8, 11, 17, 74, 76;
3:7
- sort ... 3:14, 18, 34
- stderr ... 2:32, 36
- stdin ... 2:32, 36
- stdio ... 2:29-33; 3:25
- stdout ... 2:32, 36, 59
- storage ... 11:18, 23-24, 26-28, 30,
33, 37; 12:2; 16:5; 17:3, 5-8, 16,
24-27, 36-38, 44-45, 47, 55, 58
- stream ... 2:7, 29, 30; 3:8, 10
- strip ... 2:72; 3:34
- struct ... 17:2, 29-31, 37, 62
- symbol ... 10:58, 71, 74; 11:1-3, 6,
13-14, 16-24, 26-31, 33, 35, 37-
38, 42-43, 46-47; 12:1-2, 4-7, 9,
13, 15, 17-18, 22-24, 29-30, 32
- syntax ... 12:4-5, 8-9, 32
- target ... 11:4-5; 12:1, 7
- termcap ... 10:73, 81
- terminfo ... 3:20-21; 10:1-3, 5-7, 11,
13-14, 64-66, 68-75, 77-81
- union ... 16:8; 17:2, 4, 7-8, 14, 21-22,
26-30, 34, 46, 52, 62
- window ... 10:1, 9, 12-14, 17-18, 20,
49-51, 55-58, 62, 83, 100
- word size ... 17:58
- yacc ... 2:7, 55; 3:5-8, 10-12; 5:8, 12,
14-19; 6:1-5, 8, 10-13, 15, 18-20,
22, 24-28, 32-35, 37-38, 40-42,
45, 49
- YYERROR ... 6:38, 49
- yyparse ... 6:32, 38
- yystype ... 6:40
- yytext ... 5:6-11, 14