UNIX Tool Building

Kenneth Ingham

UNIX Tool Building

Kenneth Ingham

Computer & Information Resources & Technology (CIRT)

University of New Mexico

Albuquerque, New Mexico

Academic Press, Inc.

Harcourt Brace Jovanovich, Publishers

San Diego New York Boston London

Sydney Tokyo Toronto

This book is printed on acid-free paper. ♾

The University of New Mexico holds the copyright on *watcher*. *watcher* may be freely copied, as long as the following conditions are met:

- watcher is not sold for profit
- The source code for *watcher* is made available along with the executable
- The copyright and author notices remain intact

Contact either the University of New Mexico or the author (Kenneth Ingham) if you wish to do more than this copyright allows.

Academic Press, Inc.
San Diego, California 92101

United Kingdom Edition published by
Academic Press Limited
24–28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Ingham, Kenneth, date
Unix tool building / Kenneth Ingham.
p. cm.
Includes bibliographical references.
ISBN 0-12-370830-3 (alk. paper)
1. UNIX (Computer operating system) I. Title.
QA76.76.063I54 1990
005.4'3--dc20 90-527
CIP

Printed in the United States of America
90 91 92 93 9 8 7 6 5 4 3 2 1

UNIX Tool Building

To my parents

Preface

With UNIX becoming more widespread, more people are using it to write large programs. However, much of the knowledge of how to use the tools provided with UNIX to write new tools exists in people's heads, transferred from person to person or else learned through experimentation and many frustrating days poring over the manual and wondering what the author *really* meant.

The goal of this book is to explain some of these tools in more detail as well as to introduce the reader to some of the concepts of tool building. Anyone can write programs. With a little forethought, these programs could be more general and have uses beyond just solving the one problem they were designed to solve. When thought is given to debugging before a program is written, debugging aids can be built into the program to reduce the debugging time.

Many small examples highlight specific ideas. However, as the book progresses, a large program is built to illustrate the concepts presented, tying the use of tools together and providing a unifying example. This example program makes *UNIX Tool Building* unique among books discussing programming and UNIX.

The examples used in this book worked on Ultrix 3.1, a version of UNIX based on Berkeley 4.3BSD. Output from sample commands will vary between systems.

Many people helped me write this book. Most notable were Keith Nislow who helped improve my writing skills and gave numerous comments on the style. Pat Northup helped me become consistent in my explanations, helped me aim the text at the audience I wished, and gave *many* other helpful comments. Leslie Gorsline gave many suggestions and in general has helped my writing style over the years. Finally, Diana Northup has given me much needed support over the trials of writing which I have endured. I extend my deepest appreciation to these and all of the other people who assisted me with this endeavor.

Contents

CHAPTER 9

CHAPTER 10

APPENDIX A

Introduction

1.1 Theme

The purpose of this book is to teach UNIX tool building by taking the reader through all facets of the design and implementation of a large, complete program. This book shows the reader the step-by-step development of *watcher*, a complete UNIX tool whose function is detailed later. By demonstrating new concepts in the context of this large program, rather than introducing concepts via isolated examples, the reader should get a clearer grasp of how these concepts relate to one another. By using existing UNIX tools to develop *watcher*, the reader gains a better understanding of these tools, as well as the insight that this "modular" strategy reduces the amount of time spent coding and debugging. This approach not only results in new useful tools, it also reinforces the concepts involved in their construction.

1.2 Tools

Within this book, a *tool* is a program, subroutine, or shell script which solves a general but well-defined problem. A tool differs from an ordinary program, subroutine, or shell script in its generality. It can often be reused in other contexts with little or no modification.

Consider the following example: You need to watch several computers more or less simultaneously, with the goal of averting problems before they occur. One possible solution would be to connect one terminal to each computer and move between the terminals continually, watching each system. Obviously, this arrangement wastes both terminals and time. To save time you could automate the process by creating a list of areas to watch and having each machine send you

the status of each item via electronic mail. This solution is better, since it requires only a terminal at which to read your mail. Unfortunately, *you* are still doing most of the work.

An even better solution would be to write a program which can take a list of commands to run, along with a definition of what is "normal" or "acceptable" output for these commands, and have the program notify you only when something is wrong or "abnormal." This program can be used not only to watch an operating system, but anything for which "normal" can be defined. This is an example of a tool.

As another example, suppose you have a large program split into many files of code and you need to know from how many places the subroutine **printargs** is called in this program. No tools exist on UNIX specifically designed for this purpose; however, the tool *egrep* searches files for a pattern. Another tool, *wc*, counts the lines, words, and characters in its input. These two tools can be combined to produce the desired information by using a powerful concept known as a *pipe*, where the output of one program serves as the input for another. A pipe is analogous to an assembly line, where the workers receive parts, perform their tasks with the parts, then send them down the line to the next worker. In the example about finding the number of times a subroutine is called, the following command line would produce the answer[1]:

```
egrep 'printargs(.*);' *.c | wc -l
```

With pipelines, there is little need for temporary files. On many operating systems, a command is run with the output going to a file. A second command is run with the file as its input. The pipeline is a much cleaner and more elegant solution to the problem.

Many tools are written under the assumption that they will be used as part of a pipeline. They read from the standard input, modify or use the data in some way, then write it to the standard output. For example, *troff* is good at formatting text, but the *troff* commands needed to produce a table are low-level and difficult to use. The tool *tbl*, on the other hand, recognizes certain constructions as tables and produces the *troff* code necessary to produce a table. It changes only the part of its input which pertains to tables, and the rest is passed through unchanged.[2]

1. Don't worry if this seems confusing—it will be covered in more detail in Chapter 8.

2. Programs which change part of their input and pass the rest unchanged are often known as filters.

Programs designed to use and be used in pipelines tend to be more general than those programs not designed with pipelines in mind; that is, they can solve a wider range of problems than ones which were not designed with pipelines in mind. For example, *egrep* can search for patterns in one or more files or in data which comes from the output of other programs. These other programs may be working with devices (such as *tar* or *cpio*) or simply files. All of this variety of input requires little special design effort; the program is designed to read data from standard input, no matter what is actually generating the data.

A tool does not need to be a program; a subroutine can also be a tool. As with a program, a subroutine written as a tool should solve a general problem. It can then be used in other parts of the program or even in other programs with little or no modification, saving both coding and debugging time.

When writing a subroutine to be a tool, it helps to keep the routine short, making it easy to understand the whole purpose of the routine and also easier to debug it. Understanding all of the pieces and how they interact is key to understanding how the whole functions; short routines greatly aid this process.

Shell scripts can also be tools. On UNIX, the standard shells are programmable; programs written in the shell's language are known as shell scripts. These scripts can be used to solve problems by using other programs or shell scripts. They need to meet the same criteria as programs to be considered tools.

1.3 Rationale for Using UNIX

In the strictest sense of the word, UNIX refers to the kernel—the central core of the operating system which controls the computer and divides the resources among the users. However, in most cases, when people refer to UNIX, they mean not only the kernel but also the tools usually provided with UNIX—tools such as *grep*, *awk*, and the C compiler. When "UNIX" is used in this book, this broader definition is intended.

UNIX makes writing programs easier by providing many tools which help in building new tools. Since it was written by programmers for programmers, it has one of the richest toolkits available. Here is a brief introduction to some of these tools, as well as other general terms used frequently in the UNIX environment. They will be covered in more detail later in the book.

awk A programming language which helps with data manipulation by doing part of the work (such as breaking the input up into fields) and allowing the programmer to easily express transformations to the data. *awk* can also be useful for prototyping ideas quickly to test their feasibility before doing a full implementation in a compiled language. *awk* was named for its authors: Alfred Aho, Peter Weinberger, and Brian Kernighan.

lex Assists in building lexical analyzers (scanners) by taking a description of how to break the input into logical units (known as tokens) and generating C code for a routine which reads the input and returns the tokens found. *lex*-generated scanners are usually easier to modify than those written by hand.

yacc Takes up where *lex* leaves off; given a grammar describing a language it generates C code for a parser for that language. Parsers generally pose problems, but *yacc* makes them easy to write and modify.

make Given a list of file dependencies, *make* rebuilds only the files which are out of date (such as object files depending on source files; if a source file is newer than its corresponding object file, *make* will recompile it). *make* saves programmer and CPU time.

shells The "outer layer" of the operating system. The user interacts with the operating system through a shell; the shell reads input from the user and executes commands. On UNIX, several shells are available. In addition to handling the redirecting of I/O to files or down pipelines, a shell is often also a programming language in its own right.

mail On UNIX, *mail* has several user interfaces available, and it can be used to communicate with people on the local machine, within a site, or across the world through a network.

USENET A loose network of computer systems—most running UNIX—that exchange electronic news and mail. There are many discussion groups "on the net," and information on a multitude of technical and nontechnical topics can be found among them. The technical groups are good places to learn more about UNIX (or any other topic).

UNIX also provides support for the programmer from the operating system itself, via system calls and library functions. The system calls provide access to the raw power of the operating system; the library functions make some of this power available without burdening the programmer with too many details. The services provided by these system calls and library functions range from doing I/O to keep-

ing track of and communicating with other processes in the system (or across a network).

With UNIX few decisions are already made for the programmer; almost anything can be changed (although it might take a little work). With this programmer's freedom of choice comes the programmer's responsibility to pass this flexibility on to the user. If the programmer makes decisions for the user, this has the effect of limiting the applications for which the program is useful; the resultant program might prove inadequate for applications unforeseen when the program was written. In this case, the program would have to be modified, or else a new program must be written to handle the new situation. The most useful tools are those which make the fewest *a priori* assumptions.

UNIX is available for more types of computers than any other operating system. It runs on everything from personal computers through the Cray supercomputers. Programs written for UNIX can usually be moved easily between machines made by different vendors.

There is much public-domain or inexpensive (often free) copyrighted software available for UNIX. This is due in part to UNIX's long history of use at universities where software is often given away after it is written. This software usually comes complete with source code, allowing the user to learn and make modifications (or fix bugs). This book follows the tradition by providing source which may be freely copied without royalties (see the copyright page for full information about the copyright on *watcher*).

1.4 Scope of the Book

Throughout this book it will be assumed that the reader is familiar enough with UNIX to know how to use one of the editors provided. Also necessary is at least an acquaintance with the C programming language and knowledge of how to use the C compiler provided with the system.

1.4.1 Background Information

On UNIX, there is the concept of standard I/O. When a shell starts any program, it has associated with it three I/O streams: input, output, and error. (These are commonly referred to as **stdin, stdout,** and **stderr,** pronounced "standard in," "standard out," and "standard error.") When a program reads without specifically referring to a file, it reads from the standard input. Similarly, when it writes without a reference to a specific file, it writes to the standard output. By default,

all three streams are connected to the controlling terminal (the terminal which initiated the process). With the shell, the input or output may be redirected to or from files (via > or <) or through a pipeline. The standard error is normally still attached to the controlling terminal even if the standard output is redirected; this allows error messages to be seen immediately, instead of disappearing down a pipeline or into a file.

As mentioned earlier, the shells on UNIX are programmable. A program written in a shell is usually called a shell script. Writing shell scripts is covered in Chapter 9.

1.4.2 Additional Resources

Since no one book can cover everything, the reader is directed to several books which provide information complementing the information this book presents. In Section A.4 of Appendix A is a bibliography listing these and all other books referenced throughout this book.

- Marc Rochkind's *Advanced UNIX Programming* goes into detail about the system calls and the concepts needed to understand them and also gives several small examples of how to use them.
- *The UNIX Programming Environment* by Brian Kernighan and Rob Pike is a good (but somewhat dated) introduction to using many of the tools that UNIX has to offer. It stresses the use of tools to solve problems whenever possible.
- The classic book on programming in C is *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of the language. It is the best reference book available for the language. Recently, a new edition was released which describes the ANSI standard for the C language. From here on (and elsewhere in the UNIX community), this book will be referred to as K&R.
- For more information about writing tools, *Software Tools* or its close relative *Software Tools in Pascal*, both by Brian Kernighan and P. J. Plauger, are unmatched.

The reader might also find some of the software available from the Free Software Foundation useful. Dedicated to the philosophy that source code should be available to all, the Foundation's products all come with source code and may be obtained directly from the Foundation or from anyone who has a copy. The products available are well written. Support is minimal, but since the source is provided, problems can be tracked by investigating the source.