The Unix Magician's Handbook

Gerard J. Holzmann°

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A review of "The Unix Programming Environment," by Brian W. Kernighan and Robert Pike, Prentice-Hall Software Series (1984), ISBN 0-13-937699-2.

Unix:Login, 1984, (publisher under the name Elizabeth Bimmler)

The Unix Wizard

Unix† is traditionally taught by 'wizards.' Every installation, and there seem to be well over 3000 now, inevitably comes with its own set of gurus where Unix freshmen can learn the art of Unix programming. Until recently, the prospective Unix programmer had to find a guru and seek to become his apprentice. After a long training the student could then become a master and as a token of his excellence would be issued the superuser password. Unix, to be sure, is not a trivial system, and as Kernighan and Pike note in the preface to their book: "as the Unix system has spread, the fraction of its users who are skilled in its application has decreased."

The Unix operating system has been acclaimed for its conciseness and structure, its portability, and perhaps more important still: for its availability. Commenting on the 6th Edition Unix, John Lions made the following, often quoted, observation: "the whole documentation is not unreasonably transportable in a student's briefcase." As Kernighan and Pike have aptly countered in their book: "this has been fixed in recent versions." Considering that the kernel of the first edition Unix was only 8k in size, on the average, the size of the system has more than doubled with every new edition issued. In comparison to other commercially available operating systems, the documentation that comes with the newer versions is still modest in size, but it certainly will not fit a student's briefcase anymore, unless, of course, it's left on the magnetic tape.

Unix was necessarily taught by wizards, simply because those wizards never wrote down their art. Now that the Unix system has become so popular the number of publications on Unix is steadily increasing, and a small set of textbooks has finally appeared. In some cases these textbooks convey little more than the information already available from the system documentation, though perhaps presented in a more friendly way. Oddly enough, the people responsible for the development of the Unix system have long hesitated in providing us with good quality textbooks on their system. "Well," said Rob Pike, "if somebody is going to do it anyway, why let outsiders do a bad job when insiders can do a bad job ...."

Inside Look at Unix

Two textbooks on the Unix system written by the 'insiders' have now appeared, and they haven't done such a bad job at all. A first book was written by Steve Bourne,‡ the author of the 7th edition Unix 'shell' (the command interpreter). Bourne's book gives an excellent overview of the system, with one of the best introductions to the C language and Unix system programming I have seen so far.

The second book, and the one we will consider more closely here, was written by Brian W. Kernighan and

° Disclaimer: Gerard J. Holzmann has recently joined Bell Labs, and is now a colleague of the authors of the book reviewed.
† Unix is a trademark of AT&T Bell Laboratories
‡ "The Unix System" by S.R. Bourne, Addison Wesley, Computer Science Series, (1983), ISBN 0-201-13791-7.

Rob Pike.
Brian Kernighan is a long-time member of Bell Labs Computer Science Research Center where Unix was born, nearly fifteen years ago. With others he is jointly responsible for many of Unix' utilities. He was there when Ken Thompson and Dennis Ritchie experimented with the first versions, and it was Brian who, paraphrasing the name 'Multics,' suggested the term 'Unix' as the final name for the system they developed.
Rob Pike is a true Unix 'wizard' who joined the Bell Labs group more recently. In the three years he has been there, he has already left an impressive trail of accomplishments, most notably the development (together with Bart Locanthi) of the popular 'Blit†' multi-window terminal, which is perhaps best described as a novel type of programmable workstation for Unix systems.

Kernighan & Pike's book is quite different in approach from the earlier book by Steve Bourne, though there definitely is some overlap in the material presented. Both books cover 'shell programming,' and 'document preparation' in some detail. Still, the book by Kernighan and Pike is clearly not aimed at the casual user of the Unix system. On the contrary, they aim for the reader who has mastered the basics and who wants to expand his or her knowledge.

This approach is perhaps best illustrated by the terse reminder at the start of their chapter on 'program development' (chapter eight, p. 234): "We are assuming that you understand C." For sure, not a very gentle introduction to the C language, but then again, the subject was covered at length in the earlier "C reference manual" by Kernighan and Ritchie. For those of us who do already understand C, though, the fifty pages that follow give an outstanding exposition of the development of a substantial and extremely useful program: an interpreter for a Basic-like language, written with the aid of the Unix compiler writing tools Lex and Yacc°.

Tools

The book by Kernighan and Pike is crowded with many small examples of useful tools. Shortly after I had circulated a draft of the book at the University where I worked my system was literally swamped with all the little 24-hour watchdog programs that students and co-workers had discovered in the text. As an example, here is the shell script for a small program 'watchfor:'

```
# watchfor: watch for someone to log in

PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: watchfor person' 1>&2
        exit 1
esac

until who | egrep "$1"
do
        sleep 60
done
```

In this tiny shell script the use of while loops, comments, search paths, case switches, duplication of file descriptors, exit status and shell arguments is illustrated. Let us briefly go through it.
A sharp sign in column one indicates the start of a comment: the shell will simply ignore everything that follows up to the end of the line. The search path is stored in the predefined shell variable 'PATH,' which stores a list of directory names, separated by colons. The listed directories are the only ones checked by the shell in an effort to locate a command given by the user. Traditionally Unix commands are stored in directories /bin and /usr/bin, so it suffices here to set the search path for this script to these two directories.
Next we find a case switch on the number of arguments passed to the script. This number is squirreled away

† The terminal will be marketed by the Teletype Corp. as the model 5620 'dot-mapped display'.
° Lex is a general tool for writing a lexical analyzer. Yacc can be used to write parser generators.

by the shell in a variable named ‘$#’. If the number of arguments is zero the script will print an error message and exit, returning an error status ‘1’. With the magic ‘1>&2’ we combine the standard output channel (by convention file descriptor 1), where the user will expect the output from the script to appear, with the standard error output channel (file descriptor 2). The script as presented does not provide for an appropriate reaction if the user erroneously types more than a single argument, though the change is in fact trivial:

```
case $# in
1)      until who | egrep "$1"
        do
                sleep 60
        done ;;
*)      echo ´Usage: watchfor person´ 1>&2
        exit 1
esac
```

This time the loop is only executed if the user gives exactly one argument, hopefully the login name of someone who is likely to login to the system. The argument itself is available in the variable ‘$1’. The condition of the loop consists of two commands: who and egrep, which are connected via a ‘pipe.’ The pipe will direct the output from the first command to the input of the second. Who will list all active users, egrep will select lines that match the pattern it is given in ‘$1’. The condition of the until statement is set by the exit status of the pipeline. Again ‘by convention’ this is the exit status of the last program in the pipe: in this case ‘egrep.’ Egrep will return ‘true’ (oddly enough this is the numerical value 0) if it found a match and ‘false’ (some value other than 0) if it didn’t, which is exactly the effect that we wanted here.

Other Examples

None of the examples in Kernighan & Pike’s book are artificial: they all turn out to be instructive, practical and, unfortunately, highly addictive. Here’s a short list of examples they discuss:

• simple aliasing functions, such as a little shell script named ‘cx’ as an abbreviation of ‘chmod +x’ to turn files into executable commands, or ‘lc’ as an abbreviation of ‘wc −l’ to count the number of lines in a file;
• a small computerized telephone directory server ‘tel’, that uses ‘grep’ to locate names and numbers in a list;
• a little C program ‘vis’ that makes non-printing characters in files visible;
• the following very original script that can be stored in a file named ‘2’, and linked to files named ‘3’, ‘4’, ‘5’, ... etc.:

```
pr −$0 −t −l1 $*
```

The first argument to ‘pr’ becomes the number of columns in which it will print its output. ‘−t’ turns off the usual page headers, and ‘−l1’ sets the page length to 1. ‘$0’ is the name of the script itself. (If you have access to a Unix system, try: ‘who | 2’ or ‘ls | 5’ and see just how useful this is.)
• straightforward implementations of simple database managing tools, ‘put’ and ‘get’, that can be used to keep track of the revisions that, for instance, popular programs incur;
• an especially elegant program named ‘bundle’ for packing and unpacking sets of text files (e.g. programs) for distribution via ordinary system ‘mail.’

All tools, and there are many more in the book itself, are examples of small, well structured scripts or programs that solve practical problems. In nearly all cases the book includes sample runs of the programs showing typical system reponses.

Unix Principles

The strength of the Unix system, which characterizes the true "Unix programming environment" can be summarized in a few points (see e.g. Kernighan & Pike, p. 131, or Bourne, p. 4-5):

• Files have no predefined format. The interpretation of a file’s contents depends entirely on the program that reads it.
• Programs are tools. Each tool should perform one clearly defined function, and it should be

optimized for that single function.
- The output from any program should be understandable as input to other programs. One should avoid fancy headers, trailers or spurious blank lines.
- Conversely, one should be able to replace the input of any program with the output of another. This rules out interrogative programs, that prompt the user for arguments. All the information a program needs to run is given on the command line. The program either runs or fails, and returns the appropriate exit status.
- If no arguments are given a program should read the standard input and write the standard output (keyboard and screen by default), so that the program can always be used as a filter. Optional arguments precede filename arguments. Filename arguments may specify additional *inputs,* but never an output unless properly protected (e.g. with a '−o' flag).

In an early version of Unix one of the disk maintenance commands (a predecessor of fsck, named 'check') violated the last principle. When called without an argument this program would faithfully read and check the default filesystem (there was only one) and list the errors on its standard output. If an argument was given, the corresponding file would be used to store the error list. The next version of the command, made for systems with more than just a single disk, expected the name of the filesystem to be checked to be specified in its first argument (an input instead of an output). One day an unfortunate user went back to a precious copy of the old system, typed:

```
check /dev/usr
```

and destroyed the user file system completely. Note that the all-important 'superblock' is one of the first blocks on disk and therefore also one of the first destroyed.

The File System

One of the good things about the book that Kernighan and Pike have written is that it gives access to information that one could obtain only from the gurus before. In very few other books you can find a succinct explanation of the implementation of the *file system* revealing the secrets of inodes and superblocks. There are chapters that explain how to use and how to *program the shell,* revealing the mysteries of metacharacters, quoting, shell variables and traps. Other chapters focus on the use of *system calls* in user programs, building up to the knowledge and insight required to make system programs that dig somewhat deeper into the bowels of, for instance, the file system.
Again, every topic discussed is illustrated with numerous small example programs, such as:
- a program 'waitfile' that waits until a file stops changing for a given period of time (say 60 sec.);
- a program that can correct typing mistakes in filenames by comparing the existing filenames to the user-specified filenames;
- an extremely useful program called 'readslow' that waits for the writer of a file to produce more output when the end of a file has been reached;
- a program called 'timeout' that sets an upper limit to the run time of an arbitrary other user program.

The book includes excellent tutorials on the use of esoteric programs such as 'sed,' 'awk,' 'lex' and 'yacc.' However, it is not meant to be a reference manual. "We feel it is more important to teach an approach and a style of use than just details," the authors write. The problem is of course that you can only illustrate an 'approach' to programming by either giving examples or by discussing the underlying principles. Some readers may feel that there is a little too much of the first and too little of the last in "The Unix Programming Environment." I, for one, regret that not more material was included on the design and study of the algorithms that have been used to built the Unix programming environment. Pattern matching tools such as 'grep', 'egrep', and 'fgrep,' for instance, are not very well understood. There is a lot of folklore around these programs, which has spurred many an interesting lunch discussion (which program is fastest for which problem and why). It would certainly have been revealing to see some of this folklore substantiated or rejected by a thorough discussion of the algorithms used. But then, we have at least something to look forward to in a next book on the Unix system. "The Unix Programming Environment" is not a book for utter beginners. For those, however, who not only want to be able to *use* Unix, but who want to be able to use it *well,* their book is an invaluable guide.