great deal of the complexity of current programs results from the lack of strong algebraic properties relating the primitive functions of the programming language. He sees a need for "program forming operations" with such properties, whose domains are themselves programs. With these, a rigorous approach might be found for defining the characteristics of new programs built from combinations of existing ones. Backus observes that a principal barrier to designing languages with such strong properties is the von Neumann architecture of most existing computers. He asserts that the detailed assignment and manipulation of storage which is required for each program make it difficult to define useful program-forming operations. The particular choices in managing storage for each of the programs to be composed would probably be inconsistent.

A set of operations for composing new programs from existing ones could have profound implications for hardware design. The hardware instructions would directly implement the rules for composing programs. Further, proposed functional programming approaches offer the possibility of better determining which tasks may proceed in parallel. This would allow better use of advances in very large scale integration, which make high degrees of multiprocessing most cost-effective. While this work is still in an early stage, it is likely to lead to one of the most significant advances in computer science in the 1980's.

Conclusions

Since active research in the software engineering area was begun in the late 1960's, much has been accomplished. Given stable requirements, it will be largely a matter of skilled effort and discipline to produce a predictable and reliable result. However, as indicated by the number of different and still unproven approaches to new programming methodology, this field is still very young. Thus it is likely that a decade hence the techniques in use today will be considered ill-structured and difficult to maintain. Consequently, because of the cumulative aspect of programming, which is economically rather than technically motivated, we seem destined to have an environment of the new coexisting with the old and the very old. It is fashionable for the practitioners of the contemporary art to criticize the ignorance and lack of discipline of their predecessors. It would be more fruitful to recognize that the new must coexist with and enhance the old. Successful techniques will be those which preserve a maximum of the value of that which has already been achieved. The challenge is to become masters of the evolution.

References and Notes

1. B. W. Boehm, in *Research Directions in Software Technology*, P. Wegner, Ed. (MIT Press, Cambridge, Mass., 1979), p. 44.
2. C. A. R. Hoare, *Commun. ACM* **24** (No. 2), 75 (1981).
3. C. J. Date, *Introduction to Database Systems* (Addison-Wesley, Reading, Mass., ed. 3, 1981).
4. M. W. Blasgen, *Science* **215**, 869 (1982).
5. W. M. Carlson and D. V. Kerner, *Data Base* **10** (No. 4), 3 (1979).
6. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, Reading, Mass., 1975).
7. E. W. Dijkstra, *Commun. ACM* **11** (No. 3), 147 (1960).
8. C. A. R. Hoare, *ibid.* **12** (No. 10), 576 (1969).
9. R. W. Floyd, *Math. Aspects Comput. Sci.* **19**, 19 (1967).
10. H. D. Mills, *Science* **195**, 1199 (1977).
11. M. A. Johnson, *Principles of Program Design* (Academic Press, New York, 1975).
12. M. E. Fagen, *IBM Syst. J.* **15** (No. 3), 182 (1976).
13. H. Remus, in *Software Engineering Environments* (North-Holland, Amsterdam, 1980), p. 267.
14. B. W. Kernighan and S. P. Morgan, *Science* **215**, 779 (1982).
15. T. A. Doluta, R. C. Haight, J. R. Mashey, *Bell Syst. Tech. J.* **6**, 2177 (1978).
16. J. N. Buxton and L. E. Druffel, in *Software Engineering Environments* (North-Holland, Amsterdam, 1980), p. 319.
17. R. A. DeMillo and R. J. Lipton, in *Software Metrics*, A. Perlis *et al.*, Eds. (MIT Press, Cambridge, Mass., 1981), p. 77.
18. R. Kowalski, *Commun. ACM* **22** (No. 7), 424 (1979).
19. M. Hammer and G. Ruth, in *Research Directions in Software Technology*, P. Wegner, Ed. (MIT Press, Cambridge, Mass., 1979), p. 767.
20. M. M. Lehman, *IBM Tech. Discl. Bull.* (1976).
21. J. Backus, *Commun. ACM* **21** (No. 8), 613 (1978).

The UNIX Operating System: A Model for Software Design

Brian W. Kernighan and Samuel P. Morgan

In the narrowest sense, UNIX is a time-sharing operating system, a program that controls the resources of a computer and allocates them among users. It permits programs to be run according to some scheduling policy, controls the peripheral devices (disks, tapes, printers, and the like) connected to the machine, and manages the long-term storage of information.

The authors are members of the Computing Science Research Center, Bell Laboratories, Murray Hill, New Jersey 07974.

Time sharing implies (i) an environment in which users access the system from terminals and (ii) a scheduling rule which switches rapidly among active users, to give each a share of the processor in turn. Time sharing makes it possible for people to interact with programs as they execute them; by contrast, "batch processing" implies a regimen in which users have no such interaction with programs.

Traditionally, operating systems have been large, complicated programs requiring years of effort to create. The operating system written by IBM for its System/360 series of computers, OS/360, required more than 5000 man-years of development effort (*1*). Also, most operating systems have been batch systems, with time-sharing capabilities grafted on after the fact (although this path is not universal).

In a broader sense a system, be it UNIX or OS/360, is often taken to include not only the central kernel that controls the hardware, but also essential utilities such as compilers, editors, command languages to control the sequencing of programs, and programs for manipulating files, printing information, and accounting for usage. A system may include not only all these programs, but also general-purpose programs developed merely to be run on the system. Examples include formatters for document preparation, routines for statistical analysis, and graphics packages.

This leads to the view that an operating system is built layer on layer, rather like an onion—a metaphor that also allows for wry jokes about tears. Where

0036-8075/82/0212-0779$01.00/0

"UNIX" or "system" occurs in this article, the context should indicate which layer of the onion is meant.

History

The history of UNIX is well covered in two papers by one of its creators, Dennis Ritchie (*2*, *3*). In brief, UNIX began with Ken Thompson's experiments on a discarded PDP-7 computer in 1969, after Bell Laboratories withdrew from the Multics project (*4*). (The name UNIX is a weak pun on Multics.) Thompson's sub-rosa system soon attracted Ritchie. By 1970 it had evolved sufficiently that management was persuaded to purchase a PDP-11/20 minicomputer, ostensibly to create a document preparation system, something that might today be called a word-processing system. When the document preparation software was delivered to its customer, the Bell Laboratories patent organization, in 1971, UNIX had already proven useful in many areas, with document preparation merely one application.

The PDP-11/20 was replaced by a PDP-11/45, and UNIX gradually spread throughout Bell Laboratories. Its greatest developmental leap took place in 1973, when it was rewritten from its original assembly language form into C, a high-level language developed by Ritchie (*5*). The fundamental structure of that system has been retained through all subsequent versions.

In 1975, the UNIX system was made available as a licensed software package by Western Electric to educational institutions for a nominal fee and to anyone for commercial use under a schedule of fees. In 1976, Ritchie and Stephen Johnson, taking advantage of the fact that the system and (by this time) all of the applications programs were written in C, moved the system to an Interdata 8/32, a machine of significantly different architecture from the PDP-11 (*6*). In an independent effort, Richard Miller moved the UNIX system to an Interdata 7/32 at the University of Wollongong in Australia (*7*). Since then, UNIX has been transported to other machines, including the DEC VAX-11/780, the Univac 1100, the IBM 370, the Amdahl 470, and several microcomputers.

UNIX has also become available from more than one supplier (*8*). At least a dozen companies furnish systems derived from UNIX and sold under sublicenses of a Western Electric license; other companies sell systems that are UNIX look-alikes, similar in function but developed independently to be free of licensing restrictions. By late 1981 there were well over 3000 UNIX systems worldwide: at least 1000 in the Bell System, close to 2000 at universities, and another 600 in commercial and government use. These numbers do not include microprocessor-based systems, where we have no estimates.

Summary. The UNIX operating system, a general-purpose time-sharing system, has, without marketing, advertising, or technical support, become widely used by universities and scientific research establishments. It is the de facto standard of comparison for such systems and has spawned a small industry of suppliers of UNIX variants and look-alikes. This article attempts to uncover the reasons for its success and to draw some lessons for the future of operating systems.

Overview of UNIX

File system. The file system is the mechanism whereby the operating system stores and retrieves information for users. It consists of a hierarchy of directories, each of which may contain information about other directories or files. Normally each user has a "home" directory, in which he or she creates files (programs, data, documents), and perhaps other directories to help organize large collections of files (Fig. 1). There are also directories of systems programs available to everyone. A UNIX file is merely a stream of bytes (characters). Users see no tracks, cylinders, blocks, or other device characteristics that typify commercial operating systems.

Command interpreter. The command interpreter, or "shell," accepts commands from the terminal and interprets them as requests to run programs. To run a program, it is sufficient to type its name. For example

```
who
```

lists the users currently logged on. The program name is simply the name of a file in the file system; if the file exists and is executable, it is executed. There is no distinction (as there often is in other systems) between a system program like who and one written by an ordinary user for private use, except that system programs reside in a known place for administrative convenience, and the shell searches there if it fails to find the program in the user's own directory. Although most users talk to the system through it, the shell is not part of the operating system; it is just another program. As we shall see, this is of some importance.

Input/output redirection. Normally, input and output for a program take place on the user's terminal, but the shell can be told to change either assignment to aim it at a file when the program is executed. The command line

```
program <in >out
```

instructs the shell to have program take its input from file in and place its output on file out; program itself is unaware of the change. On many systems redirection is impossible, or at best difficult, because programs believe that they should read or write only through the user's terminal. On UNIX, redirection is available to all programs without prearrangement because it is done by the shell.

Device files. Input and output devices are handled in the same manner as ordinary files. To print the output of program on a line printer (lpr) instead of writing it on the file out, one says

```
program <in >lpr
```

Of course the file in might also be a device—perhaps an instrument recording experimental data. Device files are read and written like ordinary disk files, except that reference to a device file activates the device and passes data to or from it by whatever protocol is appropriate. A new device is added to the system by writing a device driver (in C) to make the device look like another file.

Program connection: pipes. Consider the task of counting the number of people using the system. Two programs can cooperate to do this via a temporary file:

```
who >temp
wc <temp
```

who produces one line per logged-in user; wc ("wordcount") counts the lines, words, and characters.

One notable contribution of UNIX is the notion of a pipe, a mechanism for connecting programs. The "pipeline"

```
who | wc
```

performs the same task as in the example above, without using temporary files. The symbol | tells the shell to connect the output of the program on the left to

the input of the one on the right. Programs connected by a pipe run concurrently, with the system controlling and synchronizing the flow of data.

As a larger example, consider the task of plotting a graph from data. A typical UNIX approach might run a program

```
program <data
```

to produce the desired points. A separate program, plot, prepares sets of numbers for plotting on an appropriate graphics device, so

```
program <data | plot
```

produces a curve like that in Fig. 2A. If smoothing is necessary, a spline program is interpolated:

```
program <data | spline | plot
```

This produces a curve like that in Fig. 2B. Additional programs can be inserted to produce labels, and the final graph can be typeset by UNIX document preparation software.

Programming Environment

UNIX provides a host of useful programs and a powerful command interpreter for invoking them. Besides necessities like text editors and compilers, there are tools for day-to-day use (electronic mail, calendar, interuser communication) and for mechanizing frequent tasks (file comparison, searching, sorting, counting), and even some interesting games. There are also state-of-the-art tools for document preparation and programming language development.

Perhaps most significant is the style of program development that has resulted from being able to connect programs easily. Programs tend to focus on doing one thing only, but doing it well. Complex tasks are performed by separate but cooperating programs. Programs are designed so that their input can come from any other program, and their output is usable by other programs. No possible connection is foreclosed.

The software developed for document preparation on UNIX is interesting both in its own right and as an illustration of this style of program development. The basic tool is a text formatter called troff, which converts text and format specifications into commands to control a phototypesetter; troff, however, has no facilities for complicated special material such as mathematics or pictures, which are dealt with by separate programs that cooperate with troff. A program called eqn deals solely with mathematics. It recognizes portions of a document that are mathematical expressions and translates them into troff commands. For example

```
int sub 0 sup x dz over {1 + z sup 2}
~ = ~ tan sup -1 x
```

is converted into troff commands which typeset

$$\int_0^x \frac{dz}{1+z^2} = \tan^{-1} x$$

The eqn program operates as a troff preprocessor, so the usual sequence of operations is

```
eqn textfile | troff
```

The two programs cooperate, and each is much less complex than it would be if it tried to do the whole job.

The approach taken with eqn has proven so successful that other preprocessors have also been developed. The language and program for specifying tables is called tbl. It acts as a preprocessor for both eqn and troff. Another program, refer, converts brief citations to complete ones by searching a data base of references. For example, this article could be cited as

```
kernighan morgan science
```

The programs pic and ideal translate figure-drawing languages into troff commands that produce figures like those in this article.

To place all these facilities into one typesetting program would create unworkable complexity. As it is, however, each piece is documented and maintained separately and is independent of the internal characteristics of the others. Testing and debugging such a sequence of programs is much easier than it would be if they were all one.

In addition to the formatting programs, there are a variety of programs that help create better text in the first place. The earliest of these is spell, which detects spelling errors in a document. The first version of spell was developed in a few moments by pasting together existing programs for sorting and comparing word lists. The program has evolved much since then, but it remains a good example of how program development takes place in a tool-rich environment.

A more recent development is the Writer's Workbench family of programs, initiated by Lorinda Cherry (*9*). These programs examine a document for split infinitives, clichés, excessive use of passive voice, sexist phrases, and a variety of other flaws.

Software Development Tools

UNIX provides an especially congenial programming environment (*10*). The interfaces to the basic system capabilities of UNIX, particularly the input-

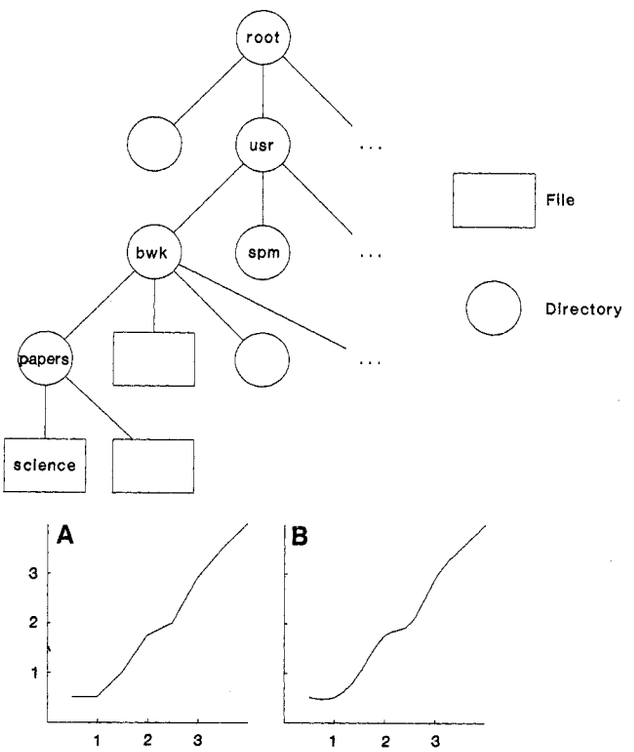

Fig. 1 (top). UNIX file system hierarchy. The directory root is the starting point for file searches through the directories of users' files and systems files. User bwk, working in his directory papers, accesses the text of this paper as science. User spm accesses the same files as /usr/bwk/papers/science. Fig. 2 (bottom). (A) Sample plot produced by the pipeline program <data | plot. (B) Sample plot produced by the pipeline program <data | spline | plot.

output system, are strikingly simple compared to those of other systems. A ten-line C program suffices to copy any file in the file system to any other. Indeed, since peripheral devices (tapes, printers, terminals, autodialers) are also treated as files, the same program can handle utility functions like tape to printer, interuser communication, and telephone calls.

Conventional programming can be avoided to a remarkable degree. The shell is an ordinary program, not part of the system kernel, so it may be invoked explicitly. This has some interesting consequences when the shell takes its input from a file instead of a terminal. If the file cmds contains commands, then

```
sh <cmds
```

runs the shell (sh) and executes the commands as if they had been typed by hand.

In fact, if a text file is marked executable, merely naming it causes the shell to execute it, so if file nu contains "who | wc" then typing the command nu counts the users. Thus the shell can be used to combine existing programs into more complicated assemblages. The resulting programs are easy to understand since they are written in a very high level language, the operations of which are entire programs. But the user of a shell program cannot, by running it, distinguish it from one written in a more conventional language.

The shell is substantially more powerful than might be inferred from such simple examples. It is a programming language in its own right, with variables, control flow, subroutines, and interrupt handling. As the shell has become more powerful, there has been a steady trend toward writing complicated sequences in the shell rather than in C. Since the search path that the shell uses to find programs can be set by each user for himself, most users have a directory of their own private commands that is searched before the normal ones. In this way, users can tailor the environment to their own preferences. The program nu to count users is a simple example of such a private program.

Many applications programs can be organized as language processors. They recognize some structured input and perform actions based on it. UNIX provides several tools for programming language development, including a compiler-compiler called YACC and a lexical-analyzer generator called LEX (*11*). The syntax of a language is specified by a grammar, with semantic actions written in C and attached to the rules of the grammar. YACC converts the grammar and actions into a parser that will process the input, executing each action when an instance of the corresponding grammatical construct occurs. Similarly, LEX converts a concise specification of the lexical tokens of a language (keywords, numbers, and so forth) into a program that will recognize them in a stream of text.

For large programs, and especially for those whose construction involves multiple processing steps such as YACC and LEX, it is convenient to have another program control the sequence of events. MAKE (*12*) accepts a specification of what to do, and does the processing steps in the right order, with minimal recompilation.

The Source Code Control System (SCCS) (*13*) was developed to deal with the problem of maintaining the consistency of numerous versions of very large programs. SCCS permits storing a history of the changes to a program throughout its lifetime, so that the program can be recreated as it was at any earlier time. The system also makes it easy to record information about why changes were made and to ensure that several programmers working on the same program do not make inconsistent changes. Although SCCS was originally intended for programs, it works just as well for managing multiple versions of manuals and other documents.

Flexibility and Ease of Change

One strength of UNIX is the degree to which it can be adapted to different requirements and environments. This is true at several levels. The use of search paths, and indeed shell programs in general, makes it possible to change the actions of commands easily. This capability is heavily used in some UNIX-based production systems developed at Bell Laboratories. The user sees something different from the standard system, but the difference is controlled by simple shell programs rather than by new programs written in C.

Any program that is not part of the system kernel can be replaced by a user with one of his own. The shell itself is the most obvious example: since it is just a user program, any user can create his own shell. Many systems have several shells in coexistence. Since the kernel itself is essentially all in C, it, too, is relatively easy to change; consequently, there are also multiple versions of the UNIX kernel.

The source code for UNIX is distributed as part of the system. Being in C, it is much easier to read, understand, and manipulate than it would be if written in assembly language. Students enjoy studying the software and then modifying it. The availability of the source code is one reason why UNIX has been successful in universities.

Of course, it is not an unmitigated blessing that the system is easy to change. One immediate result is the proliferation of variants. Mutations are necessary if evolution is to occur, but they are a nuisance in the short term.

Portability

Software written in assembly language (as most operating systems are) is forever wedded to one kind of machine. By contrast, software written in a high-level language like C is potentially portable, although care is necessary to achieve portability. Once a C compiler is available for a new machine, the UNIX software can be moved to the new environment with substantially less effort than would be required to duplicate it from scratch.

Nevertheless, transporting UNIX is not trivial. Normally it takes two or three talented people 6 months to obtain a workable production environment, but the job has been done enough times now to make UNIX available on a variety of hardware, from Amdahl 470's to Zilog Z8000's. Most users are not aware of specific hardware characteristics when running a program. Most programs are literally identical on all machines, although a few, such as compilers, have some part that is inherently machine-dependent. The system kernel itself, about 8000 lines of C, is about 95 percent identical from one machine to another.

The economic advantages of portability are great. It is highly desirable to run the same software on a variety of machines, to make use of available hardware, to avoid being tied to obsolete hardware that is no longer cost-effective, and to avoid being dependent on a single vendor.

Other Advantages

From the beginning, UNIX has been run on hardware that is popular in its own right. It is likely that UNIX would have taken longer to catch on if it had not first been available on the widely used PDP-11.

UNIX runs effectively on small machines, which makes it feasible for groups with small budgets. Furthermore, the facilities that UNIX provides—editing, text formatting, and keeping track of files—are all jobs that pervade programming, so one does not have to make radical changes in one's approach to programming to make effective use of the system. This has been particularly important for large software development projects in which the target computer was already specified. The Programmer's Workbench version of UNIX (*14*) provides a large number of tools that can be used to develop software for any computer system.

Another factor contributing to the spread of UNIX is the enthusiasm of people who are using it. For example, students who become acquainted with UNIX continue to want it when they enter industry or government.

UNIX users communicate by the telephone system and a standard set of UNIX programs for exchanging mail and files. It is not known how big this network really is, but we can readily identify more than 300 sites. As this informal network grows, there is an incentive to use UNIX to gain access to it for electronic mail.

Applications of UNIX

Text processing. UNIX programs are used for preparing the bulk of Bell Laboratories internal memoranda and manuals, patent applications, and manuscripts for publication. For technical articles the UNIX system is about twice as fast as typewriter composition (*15*). The programs have also been adopted by universities, industries, and technical societies around the world. For example, the American Physical Society has used UNIX for several years to typeset galleys for *Physical Review B*, a journal containing highly complex mathematics.

Software development. The Programmer's Workbench version of UNIX has been used inside the Bell System and outside, under license, to develop software for a large number of different computers.

Laboratory automation. Inexpensive microcomputers, acting as satellites to a standard UNIX system, control laboratory experiments and analyze and display results (*16*).

Information systems. It is easy for an individual, using standard UNIX tools such as the pattern scanning and processing language AWK (*17*), to put together programs to retrieve information from small databases; dozens of such small information systems have been made by groups using UNIX. The best-known commercial database management system based on UNIX is probably INGRES (*18*).

Computer science education. UNIX has been popular in computer science departments because of its small size and clean structure. As John Lions (*19*) of the University of New South Wales observed in 1977, "the whole documentation is not unreasonably transportable in a student's briefcase." Unfortunately, Lions's remark is less true today.

Nonapplications

Real-time systems. UNIX was designed for a time-sharing environment in which users give commands to a system that does a significant amount of computation in response to each command. It was not designed for real-time control of high-speed equipment, in which responses must be made to critical inputs in milliseconds or a strict schedule of deadlines met for critical outputs.

Large databases. Similarly, UNIX was not designed to handle large volumes of high-speed transactions, as would be generated, for example, by an airline reservations system in which individual commands require only trivial computation but do require quick access to large disk files, together with explicit provisions for consistency control and quick recovery from system shutdown. UNIX has been used for various small information management systems because UNIX-based systems are easy to maintain and modify. As of today, however, large databases have to trade flexibility for performance, and maximum performance still requires a specialized database management system on a large computer.

Nonprogrammers. The interface between UNIX and users is by no means as elegant as the underlying system design (*20*). Professional programmers (like Thompson and Ritchie) accordingly tend to be much more enthusiastic about the beauties of UNIX than casual users or nonprogrammers. The tools to build a smooth interface between users and, say, a UNIX-based personal computing system or office automation system certainly exist, and various entrepreneurs already provide interfaces more suited to nonspecialists. But the standard version of UNIX is reminiscent of the Model T Ford in that users are expected to customize it themselves.

Conclusions

UNIX is by no means the end of the road in operating systems, but there are some technical lessons in its success for designers of future operating systems and other software. UNIX demonstrates that the right combination of ideas implemented straightforwardly can be remarkably effective. A simple file system is much easier to build than the traditional commercial ones and more convenient to use. A separate command interpreter is an excellent way to organize command execution in a time-sharing system. Program interconnection not only makes it easier to write and use programs, but seems to foster good design and a tool-building attitude among its users. High-level languages are here to stay; they extract a moderate cost in space and time, but pay off in comprehensibility, ease of change, and portability. Finally, the fact that UNIX was developed literally in an attic by two people indicates that there is still a place for individual contributions to software. A good product can find its way without marketing; indeed it may be the better for having no marketing concerns to drive it.

References and Notes

1. F. P. Brooks, Jr., *The Mythical Man-Month* (Addison-Wesley, Reading, Mass., 1975), p. 31.
2. D. M. Ritchie, *Bell Syst. Tech. J.* **57**, 1947 (1978).
3. ———, paper presented at the Symposium on Language Design and Programming Methodology, Sydney, September 1979.
4. E. I. Organick, *The MULTICS System* (MIT Press, Cambridge, Mass., 1972).
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice-Hall, Englewood Cliffs, N.J., 1978).
6. S. C. Johnson and D. M. Ritchie, *Bell Syst. Tech. J.* **57**, 2021 (1978).
7. R. Miller, *Oper. Syst. Rev.* **12** (No. 3), 32 (1978).
8. R. C. Johnson, *Electronics* **54**, 119 (1981).
9. L. L. Cherry, paper presented at Association for Computing Machinery Symposium on Text Manipulation, Portland, Ore., June 1981.
10. B. W. Kernighan and J. R. Mashey, *Software Pract. Exper.* **9**, 1 (1979).
11. S. C. Johnson and M. E. Lesk, *Bell Syst. Tech. J.* **57**, 2155 (1978).
12. S. I. Feldman, *Software Pract. Exper.* **9**, 255 (1979).
13. A. L. Glasser, paper presented at the Association for Computing Machinery Software Quality Assurance Workshop, San Diego, Calif., 15 to 17 November 1978.
14. T. A. Dolotta, R. C. Haight, J. R. Mashey, *Bell Syst. Tech. J.* **57**, 2177 (1978).
15. B. W. Kernighan, M. E. Lesk, J. F. Ossanna, *ibid.*, p. 2115.
16. B. C. Wonsiewiecz, A. R. Storm, J. D. Sieber, *ibid.*, p. 2209.
17. A. V. Aho, P. J. Weinberger, B. W. Kernighan, *Software Pract. Exp.* **9**, 267 (1979).
18. M. Stonebraker, *Assoc. Comput. Mach. Trans. Database Syst.* **5**, 412 (1980).
19. J. Lions, personal communication.
20. D. A. Norman, *Datamation* **27** (No. 12), 139 (1981).