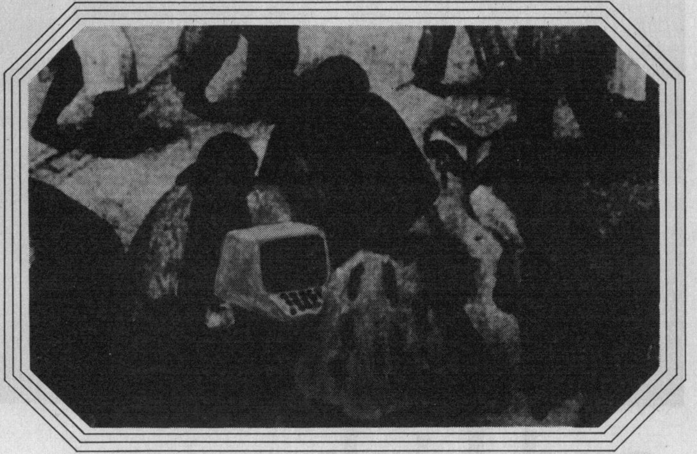
Since complex Unix tools are built from simple, single-function components, programmers see their work as the creation and use of tools. This view encourages growth, not reinvention.

The Unix Programming Environment

Brian W. Kernighan
Bell Laboratories, Murray Hill, N.J.

John R. Mashey
Bell Laboratories, Whippany, N.J.



"Software stands between the user and the machine."
—Harlan D. Mills

There is more than a grain of truth in this remark. Many operating systems do some things well, but seem to spend a substantial fraction of their resources interfering with their users. They are often clumsy and awkward, presenting major obstacles to getting a job done.

Things needn't be that way. For over eight years, we have used the Unix* operating system[1] and have found it helpful, productive, and a pleasure to use.

We are not the only ones who feel this way. Although the basic Unix system was literally developed in a year by two people working in an attic, and has, until recently, been available only as an unsupported package, the benefits it provides are so compelling that over 2500 Unix systems are now in place around the world. At Bell Laboratories, Unix systems provide more timesharing ports than all other systems combined. These ports are accessed by thousands of people; many use them on a daily basis. Unix has spawned a host of offshoots—at least six companies† offer or plan to offer systems derived from or compatible with the Unix system, for processors ranging from microprocessors to large mainframes.

In this article we describe what appears to be a new way of computing. We emphasize those things that are unique, particularly well done, or especially good for productivity. We also discuss aspects of the system that have changed our view of the programming process itself and draw some lessons that may be valuable to future implementors of operating systems.

Neither of us was involved with the development of the Unix system, although we have contributed applications software. We describe the system from the user's viewpoint, based on our own experiences and those of the large community of users with whom we have been involved. This is a valid perspective because good systems have many more users than developers. (A developer's retrospective can be found in Ritchie.[2])

File system and input/output

File system structure. As any operating system should, Unix provides facilities for running programs and a file system for managing information. The basic structure of the file system is fairly conventional—there is a rooted tree in which each interior node is a directory (that is, a list of files and directories), and each leaf is either a file or a directory (see Figure 1). Any file can be accessed by its name, either relative to the current directory or by a full path name that specifies its absolute position in the hierarchy. Users can change their current directory to any position in the hierarchy. A protection mechanism prevents unauthorized access to files.

Several design choices increase the uniformity of the file system by minimizing irrelevant distinctions and arbitrary special cases. These choices permit programs that access the file system to be substantially simpler and smaller than they would be if this regularity were absent.

First, *directories are files*. The only distinction between a directory and an ordinary file is that the system reserves to itself the right to alter the contents of a directory. This

*Unix is a trademark of Bell Laboratories.
†Cromemco, Onyx, Yourdon, Whitesmiths, Amdahl, and Wollongong Group.

This paper is adapted from "The UNIX Programming Environment," by Brian W. Kernighan and John R. Mashey, originally published in *Software—Practice & Experience,* Vol. 9, No. 1, Jan. 1979, and reproduced by permission of the editors and the publishers, John Wiley & Sons.

is necessary because directories contain information about the physical structure of the file system. Since directories are files, they can be read (subject to the normal permission mechanism) just as ordinary files can. This implies that programs such as the directory lister are in no sense special. They read information that has a particular format, but they are not system programs.

In many systems, programs like directory listers are believed to be (and often are) part of the operating system. In the Unix system, they are not. One of the distinguishing characteristics of Unix is the degree to which this and similar "system" functions are implemented as ordinary user programs. This approach has significant benefits: it reduces the number of programs that must be maintained by system programmers, it makes modification easier and safer, and it increases the probability that a dissatisfied user will rewrite (and perhaps improve) the program.

The next aspect of the file system is critical: *a file is just a sequence of bytes.* As far as the file system is concerned, a file has no internal structure; it is a featureless, contiguous array of bytes. In fact, a file is better described by the attributes it lacks.

- There are no tracks or cylinders; the system conceals the physical characteristics of devices instead of flaunting them.
- There are no physical or logical records or associated counts; the only bytes in a file are the ones put there by the user.
- Since there are no records, there is no fixed/variable length distinction and no blocking.
- There is no preallocation of file space; a file is as big as it needs to be. If another byte is written at the end of a file, the file is one byte bigger.
- There is no distinction between random and sequential access; the bytes of a file are accessible in any order.
- There are neither file types for different kinds of data nor any access methods; all files are identical in form.
- There is no user-controlled buffering; the system buffers all I/O itself.

Although these may seem like grave deficiencies, in fact they are major contributions to the effectiveness of the system. The file system strives to hide the idiosyncrasies of particular devices upon which files reside, so all files can look alike.

It should not be inferred from the foregoing that files do not have structure. Certain programs do write data in particular forms for the benefit of people or other programs. For example, the assembler creates object files in the form expected by the loader, the system itself uses a well-defined layout for the contents of a directory, and most programs that manipulate textual information treat it as a stream of characters with each line terminated by a newline character. But these structures are imposed by the programs, not by the operating system.

Programming interface. Seven functions comprise the programmer's primary interface to the file system: OPEN, CREATE, READ, WRITE, SEEK, CLOSE, and UNLINK. These functions are direct entries into the operating system.

To access a file, OPEN or CREATE must be used:

```
FD = OPEN(FILENAME, MODE)
FD = CREATE(FILENAME, MODE)
```

OPEN opens FILENAME for reading, writing, or both, depending on mode. FILENAME is simply the name of the file in the file system—a string of characters. CREATE also opens a file, but truncates it to zero length for rewriting, in case it already exists. It does not complain if the file already exists.

Both OPEN and CREATE return a file descriptor, a small positive integer that serves thereafter as the connection between the file and I/O calls in the program. (A negative return indicates an error of some sort.) The file descriptor is the only connection; there are no data control blocks in the user's address space.

Actual input and output are done with READ and WRITE.

```
N_RECEIVED = READ(FD, BUF, N)
N_WRITTEN = WRITE(FD, BUF, N)
```

Both calls request the transfer of N bytes to or from the buffer BUF in the user's program from or to the file specified by FD; N can have any positive value.

Both READ and WRITE return the number of bytes actually transferred. This can be less than N, when, for example, the system reads a file whose size is not a multiple of N bytes. A return of zero on reading signals the end of file.

As far as a user program is concerned, input and output are synchronous and take place in chunks of whatever size is requested. The system handles buffering and blocking into proper sizes for physical devices. Not only does this simplify user programs, it converts the haphazard suboptimizations of individual user programs into global optimization across the entire set of active programs. For example, the system handles queues of disk requests so that disk head motion and rotational delays are minimized. In many applications, this approach actually improves disk performance, which is often more critical than CPU performance. Such global optimization aids adaptation to both changes in disk configuration and the increasingly common use of larger, but fewer, disks per system.

I/O is normally sequential, that is, each command continues where the preceding one left off. This default may be changed by the call SEEK.

```
SEEK(FD, POSITION, RELATIVE_TO)
```

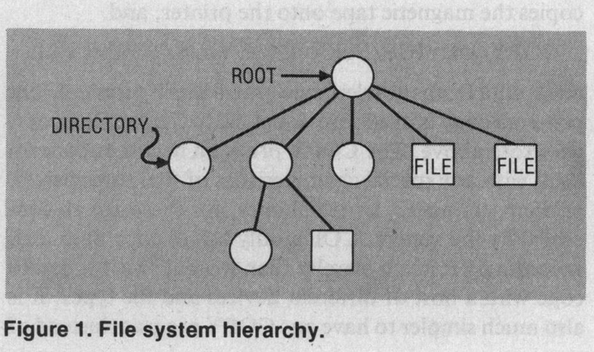


Figure 1. File system hierarchy.

This requests that the pointer for the next read or write be set to the byte specified by POSITION, relative to the beginning, current position, or end as specified by RELATIVE_TO. Thus, SEEK provides a convenient random access capability.

Finally, the function CLOSE(FD) breaks the connection between a file descriptor and an open file; UNLINK(NAME) removes the file from the file system.

Given this interface, many programs become simple indeed. For example, here is the executable part of a program COPY that copies one file to another, written in the C programming language.[3,4]

```
FIN = OPEN(NAME1, READMODE);
FOUT = CREATE(NAME2, WRITEMODE);
WHILE ((N = READ(FIN, BUF, SIZEOF BUF) > 0)
        WRITE(FOUT, BUF, N);
```

The buffer BUF may be of any convenient size. The file names NAME1 and NAME2 are character strings, typically set from the command line when the program is executed. Another half-dozen lines of declarations make this into a complete program that will copy any file to any other file.

Input/output devices. The interface described above applies to all files. This goes further than might be expected, for all peripheral devices are also files in the file system. Disks, tapes, terminals, communications links, the memory, and the telephone system all have entries in the file system. When a program tries to open one, however, the system brings into execution the proper driver for the device, and subsequent I/O goes through that driver. The I/O device files all reside in one directory for convenient administration, and they can be distinguished from ordinary files by the rare programs that need to do so. In general, however, considerations specific to particular devices are pushed out into the device drivers where they belong, and user programs need know nothing about them. The file system conceals the physical peculiarities of devices instead of making them visible.

From the programmer's standpoint, the homogeneity of files and peripheral devices is a considerable simplification. For example, the file copy program COPY that we wrote in the previous section could be invoked as

```
COPY FILE1 FILE2
```

to copy the contents of FILE1 to FILE2. But the files can be devices, so

```
COPY /DEVICE/TAPE /DEVICE/PRINTER
```

copies the magnetic tape onto the printer, and

```
COPY/DEVICE/PHONE/DEVICE/TERMINAL
```

reads data from the telephone onto a user's terminal. The program copy is in all cases identical to the four lines of C we wrote above. The COPY program need not concern itself with any special characteristics of files, tape drives, printers, terminals, or telephones, for these are all concealed by the system. COPY only has to copy data, and accordingly is much simpler than it would be if it had to cope with a host of different devices and file types. It is also much simpler to have one COPY program instead of a host of different "utility" programs corresponding to the host of different possible copying operations.

As another instance of the value of integrating I/O devices into the file system, interuser communication by the WRITE command is trivial. Since a user's terminal is a file, no special mechanism is needed to write on it. Unwanted messages can be prevented merely by changing the permissions on the terminal, to make it impossible for others to write on it.

Simplicity is achieved by the elimination of special cases, such as discrimination between devices and files.

The user interface

Running programs. When a user logs into a Unix system, a command interpreter called the shell[5,6] accepts commands from the terminal and interprets them as requests to run programs. The form is as suggested above: a program name, perhaps followed by a list of blank-separated arguments that are made available to the program. For example, the command

```
DATE
```

prints

```
WED OCT 29 09:45:24 EST 1980
```

The program name is simply the name of a file in the file system; if the file exists and is executable, it is loaded as a program. There is no distinction between a "system" program like DATE and one written by an ordinary user for private use, except that system programs reside in a known place for administrative convenience. Commonly used programs such as DATE are kept in one or two directories, and the shell searches these directories if it fails to find the program in the user's own directory. (It is even possible to replace the shell's default search path with one's own.) Installing a new program requires only copying it into this directory:

```
COPY COPY /COMMAND/COPY
```

installs copy from the current directory as the new system version in /COMMAND.

Filename shorthand. A typical Unix system lives and breathes with file system activity. Most users tend to have a large number of small files; the Bell Labs system used for computing science research, for example, has about 45,000 files for about 50 active users; the average file size is about 10,000 bytes, but the median is much smaller.

Most programs accept a list of file names as parameters; lists are often quite long. For example, here is a listing of a directory.

```
ADDSET.C                        TEMP1
COMMON                          TEMP2
DODASH.C                        TEMP3
ESC.C                           TEMP4
FILSET.C                        TEMP5
GETCODE                         TRANSLIT.C
MAKSET.C                        XINDEX.C
TEMP                            XLATE.A
```

The names that end in .C are C source programs (a convention, not a requirement of the operating system). To print all these files with the command PR, one could say

```
PR ADDSET.C DODASH.C ESC.C FILSET.C
        MAKSET.C TRANSLIT.C XINDEX.C
```

but this is obviously a nuisance and impossible to get right the first time. The shell, however, provides a shorthand. In the command

```
PR *.C
```

the character * is interpreted by the shell as "match anything." The current directory is searched for names that (in this case) end in .C, and the expanded list of names is handed to PR; PR is unaware of the expansion.

The shell also recognizes other pattern-matching characters, less frequently used than *. For example,

```
RM TEMP[1-5]
```

removes TEMP1 through TEMP5 but does not touch TEMP.

Filename shorthand is invaluable. It greatly reduces the number of errors in which a long list is botched or a name omitted, it encourages systematic naming of files, and it makes it possible to process sets of files as easily as single ones. Incorporating the mechanism into the shell is more efficient than duplicating it everywhere and ensures that it is available to all programs in a uniform way.

Input/output redirection. As we mentioned earlier, the user's terminal is just another file in the file system. Terminal I/O is so common, however, that by convention the command interpreter opens file descriptors 0 and 1 for reading and writing the user's terminal before executing a program. In this way, a program that intends only to read and write the terminal need not use OPEN or CLOSE.

The command interpreter can also be instructed to change the assignment of input or output to a file before executing a program.

```
PROGRAM <IN >OUT
```

instructs the shell to arrange that PROGRAM take its input from IN and place its output on OUT; PROGRAM itself is unaware of the change.

The program LS produces a listing of files in the current directory, redirecting the output with

```
LS >FILELIST
```

which collects the list in a file. The program WHO prints a list of currently logged-on users, one per line.

```
WHO >USERLIST
```

produces the same list in the file USERLIST. If the file named after > exists, it is overwritten, but it is also possible to append instead of replace:

```
WHO >>USERLIST
```

appends the new information to the end of USERLIST.

The text editor is called E;

```
E <SCRIPT
```

runs it from a script of previously prepared editing commands.

These examples have been chosen advisedly. On many systems, this set of operations is impossible because in each case the corresponding program firmly believes that it should read or write the terminal, and there is no way to alter this assumption. On other systems, it is possible, but difficult, to perform the redirection. It is not enough, however, for a procedure to be just barely possible; it must be easy. The < and > notation is easy and natural.

Again, observe that the facility is provided by the command interpreter, not by individual programs. In this way, it is universally available without prearrangement.

Tools. One of the most productive aspects of the Unix environment is its provision of a rich set of small, generally useful programs—tools—for helping with day-to-day computing tasks. The programs shown below are among the more useful. We will use them to illustrate other points in later sections of the article.

WC FILES . . .
  Count lines, words, and characters in files.

PR FILES . . .
  Print files with headings, multiple columns, etc.

LPR FILES . . .
  Spool files onto line printer.

GREP PATTERN FILES . . .
  Print all lines containing pattern.

Much of any programmer's work is merely running these and related programs. For example,

```
WC *.C
```

counts a set of C source files;

```
GREP GOTO *.C
```

finds all the GOTOs.

Program connection. Suppose we want to count the number of file names produced by the LS command. Rather than counting by hand or modifying LS to produce a count, we can use two existing programs in combination.

```
LS >FILELIST
WC <FILELIST
```

LS produces one line per file; WC counts the lines.

As another example, consider preparing a multicolumn list of the file names on the on-line printer. We use the multicolumn capabilities of the PR command and the spooling provided by LPR.

```
LS >FILELIST
PR -4 <FILELIST >TEMP
LPR <TEMP
```

This is an example of separation of function, one of the most characteristic features of Unix usage. Rather than combining things into one big program that does everything (and probably not too well), one uses separate programs, temporarily connected as needed.

Each program is specialized to one task and accordingly is simpler than it would be if it attempted more. It is unlikely that a directory-listing program could print in multiple columns, and to ask it to also spool for a line printer would be preposterous. Yet the combination of operations is obviously useful, and the natural way to achieve it is by a series connection of three programs.

Pipes. It seems silly to have to use temporary files just to capture the output of one program and direct it into the input of another. The Unix pipe facility performs exactly this series connection without any need for a temporary file. The pipeline

```
LS | PR -4 | LPR
```

is a command line that performs the same task as the example above. The symbol | tells the shell to create a pipe that connects the standard output of the program on the left to the standard input of the program on the right. Programs connected by a pipe run concurrently, with the system taking care of buffering and synchronization. The programs themselves are oblivious to the I/O redirection. The syntax is again concise and natural; pipes are readily taught to nonprogramming users.

In principle, the pipe notation could be merely a shorthand for the longer form with temporaries. There are, however, significant advantages in running the processes concurrently, with hidden buffers instead of files serving as the data channels. A pipe is not limited to a maximum file size and therefore can cope with an arbitrary amount of data. Also, output from the last command can reach the terminal before the first command receives all of its input—a valuable property when the first command is an interactive program like a desk calculator or editor.

As a rule, most programs neither know nor care that their input or output is associated with a terminal, a file, or a pipe. Commands can be written in the simplest possible way, yet used in a variety of contexts without prearrangement. This would be much less possible if files did not share a common format.

As an example of a production use of program connection, a major application on many Unix systems is document preparation. Three or four separate programs are used to prepare typical documents: TROFF, the basic formatting program that drives a typesetter; EQN, a preprocessor for TROFF that deals solely with describing mathematical expressions; TBL, a table-formatting program that acts as a preprocessor for both EQN and TROFF; REFER, a program that converts brief citations to complete ones by searching a data base of bibliographic references; PIC, a program that translates a language into commands for drawing simple figures; and a number of postprocessors for TROFF that produce output on various devices. Placing all of these facilities into one typesetting language and program not only would create an absolutely unworkable monster, it would not fit into the limited address space of the PDP-11. As it is, however, each piece is independent enough to be documented and maintained entirely separately. Each is independent of the internal characteristics of the others. Testing and debugging such a sequence of programs is immensely easier than it would be if they were all one, simply because the intermediate states are clearly visible and can be materialized in files at any time.

Since programs can interact, novel interactions spring up. Consider three programs: WHO, which lists the currently logged-on users, one per line; GREP, which searches its input for all occurrences of lines containing a particular pattern; and WC, which counts the lines, words, and characters in its input. Taken individually, each is a useful tool. But consider some combinations:

```
WHO | GREP JOE
```

tells whether JOE is presently logged in,

```
WHO | WC
```

tells how many people are logged in, and

```
WHO | GREP JOE | WC
```

tells how many times JOE is logged in. None of these services requires any programming, just the combination of existing parts.

The knowledge that a program might be a component in a pipeline enforces a certain discipline on its properties. Most programs read and write the standard input and output if it is at all sensible to do so; accordingly, it is easy to investigate their properties by typing at them and watching their responses. Programs tend to have few encrustations and features (WHO will not count its users, or tell you that JOE is logged on). Instead, they concentrate on doing one thing well, and they are designed to interact with other programs; the system provides an easy and elegant way to make the connection. The interconnections are limited not by preconceptions built into the system, but by the users' imaginations.

In this environment, people begin to search for ways to use existing tools instead of laboriously making new ones from scratch. As a trivial example, a colleague needed a rhyming dictionary, sorted so that words ending in "a" come before those ending in "b," and so on. Instead of writing a special SORT or modifying the existing one, he wrote the trivial program REV, which reverses each line of its input. Then

```
REV <DICT | SORT | REV >RHYMINGDICT
```

does the job. Note that REV need only read and write the standard input and output.

Placing a sorting program in a pipeline illustrates another element of design. The pipe notation is so natural that it is well worthwhile to package programs as pipeline elements ("filters") even when, like SORT, they can't actually produce any output until all their input is processed. Recall the uses of GREP: it has appeared as the source for a pipeline, as the sink, and in the middle.

The existence of pipes encourages new designs as well as new connections. For example, a derivative of the editor, called a stream editor, is often used in pipelines, and the shell may well read a stream of dynamically generated commands from a pipe.

Program sizes. The fact that so many tasks can be performed by assemblages of existing programs, perhaps augmented by simple new ones, has led to an interesting phenomenon—the average Unix program is rather small, measured in lines of source code.

Figure 2 demonstrates this vividly. The number of lines of source in 106 programs, including most of the commonly used commands, but excluding compilers, was counted with WC. The counts have been sorted into increasing order of number of source lines with SORT, converted into a graph with GRAPH, then converted into TROFF commands. The *x* axis is the number of lines; the *y* axis is simply the ordinal number of the program.

The median program here is about 250 lines long; the 90th percentile is at about 1200 lines. That is, 90 percent of these programs have less than 1200 lines of source code (about 20 pages). Clearly, it is much easier to deal with a 20-page program than a 100-page program.

The programs are written in C, as are essentially all Unix programs that yield executable code, including the operating system itself. We feel that C itself is another source of high productivity—it is an expressive and versatile language, yet efficient enough that there is no compulsion to write assembly language for even the most critical applications.

C is available on a wide variety of machines, and with only modest effort it is possible to write C programs that are *portable,* programs that will compile and run without change on other machines. It is now routine in our environment for programs developed for the Unix system to be exported unchanged to other systems. There is obviously a considerable gain in productivity in not having to rewrite the same program for each new machine.

Since the operating system itself and all of its software is written in C, it too is portable. The Unix system itself has been moved from the PDP-11 to, among others, the Interdata 7/32 and 8/32, DEC VAX 11/780, Univac 1100, Amdahl 470/V7, and IBM S/370. From the user's standpoint, these systems are indistinguishable to the point that a command called WHERE, which identifies the current machine, has become widely popular. The original porting of Unix to the Interdata 8/32 is described by Johnson and Ritchie[7]; Miller describes transporting Unix to the Interdata 7/32 in an independent experiment.[8] There are also Unix lookalike systems on a variety of microcomputers.

Avoiding programming

The command language. We have already mentioned the basic capabilities of the Unix shell, which serves as the command interpreter. The critical point is that it is an ordinary user program, not a part of the system.[5] This has several implications: the shell can readily evolve to meet changing requirements and can be replaced by special versions for special purposes on a per-user basis. Perhaps most important, it can be made quite powerful without consuming valuable system space.

Much of the use of the shell is simply to avoid programming. The shell is an ordinary program, so its input can be redirected with <. Thus, if a set of commands are placed in a file, they can be executed just as if they had been typed. The command to do so is

```
SH <CMDFILE
```

(SH is the name of the shell). The file CMDFILE has no special properties or format—it is merely whatever would have been typed on the terminal, but placed in a file instead. Thus, a "catalogued procedure" facility is not a special case, but a natural by-product of the standard I/O mechanism.

This is such a useful capability that several steps have been taken to make it even more valuable. The first is the addition of a limited macro capability. If there are arguments on the command line that invoke the procedure, they are available within the shell procedure

```
SH CMDFILE ARG1 ARG2 . . .
```

It is manifestly a nuisance to have to type SH to run such a sequence of commands; it also creates an artificial distinction between different kinds of programs. Thus, if a file is marked executable but contains text, it is assumed to be a shell procedure and can be run by

```
CMDFILE ARG1 ARG2 . . .
```

In this way, CMDFILE becomes indistinguishable from a program written in a conventional language; syntactically and semantically, the user sees no difference whatsoever between a program that has been written in hard code and one that is a shell procedure. This is desirable not only for ease of use, but because the implementation of a given command can be changed without affecting anyone.

As a simple example, consider the shell program TEL, which uses GREP to search an ordinary text file, /USR/LIB/TEL, for telephone numbers, names, etc. In its entirety, the procedure is

```
GREP $1 /USR/LIB/TEL
```

$1 stands for the first argument when the command is called; the command

```
TEL BWK
```

produces

```
BRIAN KERNIGHAN (BWK) 6021
```

Since TEL uses the general-purpose pattern finder GREP, not a special program that knows only about telephone directories, the commands TEL 6021, TEL BRIAN, and TEL KERN all produce the same entry.

The shell is actually substantially more powerful than might be inferred from simple examples like TEL. It is a

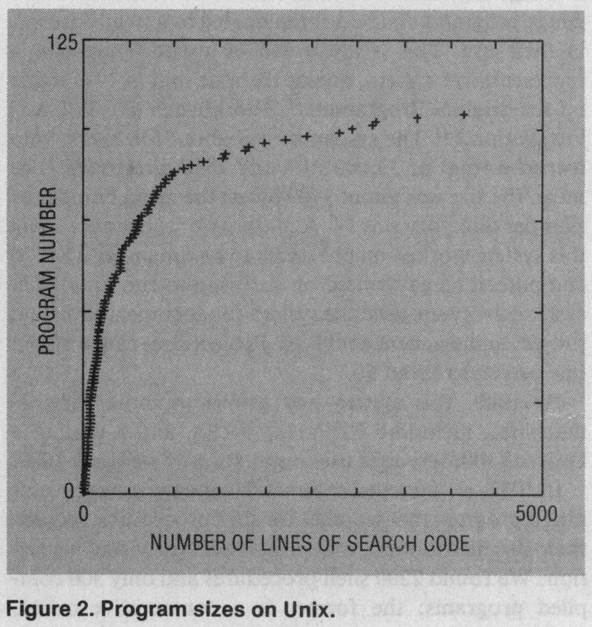


Figure 2. Program sizes on Unix.

programming language in its own right, with variables, control flow, subroutines (calling other programs), and even interrupt handling. In fact, as the shell has become more powerful and provided more facilities, there has been a steady trend toward writing complicated processes in the shell instead of in C. However, it has remained true that the shell is just an ordinary user program; its input is ordinary text, and a user cannot, by running a program, determine whether or not it is a shell process.

The shell is substantially more powerful than might be inferred from simple examples like TEL. It is a programming language in its own right.

Although the shell resembles a typical procedural language, it has rather different qualities. Most important, in certain ways it is a very high-level language, and as such is far easier to learn, use, and understand than lower-level languages. Shell programs are inherently easier to understand and modify than conventional programs because they are small (usually a handful of lines) and use familiar high-level building blocks.

The shell language is rapidly extensible—users can create new commands on the spur of the moment, and it can be adapted to meet performance requirements without disturbing its user interface. The elements of its language are generally quite independent, that is, changes to most pieces can be made without affecting the others. The shell provides most of the interconnection among programs—the complexity of interaction is linear (or less) because components are so independent of one another. As a result, it is difficult, even for a beginner, to write unmodular shell procedures. Modularity is inherent in the language and occurs without effort or careful planning. The fact that the language has no GOTO statement probably helps.

Usage statistics. The ease with which command language programs can be written has led to a steady growth in their use. This is illustrated by usage figures for a representative system, one of the nine that in 1977 made up the original Programmer's Workbench (PWB/Unix) installation.[9-11] The system served about 350 users, who owned a total of 39,000 files and 2850 directories. The mean file size was about 3700 bytes; the mean number of files per directory was 14. A majority of the people using this system worked on programs to be run on IBM S/370 computers; some worked on software to run on a Unix system; everyone used the system for documentation and project management activities. Project sizes ranged from one person to about 50.

By 1980, this system had grown to more than 20 machines, including PDP-11s, VAXs, and a part of a Univac 1100, serving a user population of well over 1000.

In 1977, we surveyed command language usage by running a program that searches for shell procedures, records their size distribution, and prints them for visual inspection. We found 2200 shell procedures and only 500 compiled programs; the former is a conservative count, because the search program necessarily misses some files that actually are shell procedures. The shell procedures counted were fairly small, averaging a little over 700 bytes apiece. Examining the distribution of lines per procedure, we found a mean of 29 lines, a median of 12, and a mode of one. In fact, 11.7 percent of all procedures consisted of but a single line. About 45-50 percent of the procedures contained some conditional logic; about half of these (or 20-25 percent of the total) included loops, primarily to perform the same operation on each file in an argument list.

Spot checks in 1980 indicate that programming usage increased since 1977, as it had between 1975 and 1977.[12,13] The proportion of shell programs is higher in the more recent period, though the characteristics of individual programs are much the same. One new wrinkle is that it has become common for each user to have a collection of personal commands, a result of the fact that the shell permits users to alter the default search path for finding commands. These personal commands are almost invariably shell programs.

Several conclusions can be drawn. First, people make significant use of shell procedures to customize the general environment to their particular needs, if only to abbreviate straight-line sequences of commands. For example, most one-line procedures consist of a single command (like TEL) or pipeline and are often used to provide fixed argument values to commands that cannot reasonably know correct default values. Thus, commands need not be complicated by special default rules, but can still be quickly customized for local needs. Second, programming goals are accomplished by writing shell procedures rather than compiled programs. Examples include small data base management packages, procedures to generate complex job control language for other systems, and project management procedures for configuration control, system regeneration, project scheduling, data dictionary management, and interuser communication. Third, as people become accustomed to this methodology, its use increases with time.

Current programming methodology. An unusual programming methodology grows from the combination of a good toolkit of reliable programs that work together, a command language with strong programming features, and the need to manage constant change at reasonable cost.

First, it is often possible to avoid programming completely because some combination of parts from the toolkit can do the job. A spectrum of cooperating utilities like GREP, SORT, and WC goes a long way toward handling many of the simple tasks that occur every programming day. In addition, we are seeing the development of general-purpose data transformers that can convert data from a file or program into some different form for another program. One notable example is SED, the stream-oriented version of the text editor.

Second, if a program is necessary, the initial version can often be written as a shell procedure instead of as a C program. This approach permits a prototype to be built quickly, with minimal investment of time and effort. If it is used a few times and thrown away, no great effort has been expended. Even if a C program is needed, it may well be tiny, performing some simple transformation like the REV program.

Third, almost any program must be continually modified to meet changing requirements, and no amount of initial design work is a complete substitute for actual use. In fact, too much design without experience can lead to a first-class solution to the wrong problem. A program may require a period of rapid and drastic evolution before stabilizing. Modification of a shell procedure is both cheap and reliable, since it is a small object built of generally reliable parts and exists only as a file of editable text. No compilation is necessary, and there are no object modules to maintain and update.

Fourth, once a procedure evolves to an effective, more-or-less stable state, it can be left alone—if it is fast enough for its intended uses, which are by then well known. If it is too slow, it can be entirely rewritten in C, or at least some small, crucial section can be recoded, with the existing version providing a proven functional specification. Deferral of efficiency considerations until the design and the usage patterns have stabilized usually prevents the all too common error of premature optimization.

Capabilities are improved in several ways. A task that recurs frequently may show the clear need for a general-purpose tool; by the time it is written, its requirements are fairly well defined. An existing tool can be upgraded as it is recognized that some change would enhance its usability or performance. Finally, new ways of combining programs can be added to the shell.

The effect of this methodology is to substitute reliable, low-cost programming for unreliable or expensive programming. The effort required to produce both reliability and efficiency is reserved for code that really requires these attributes. Effort is applied efficiently because accurate requirements are known by the time the code is written.

Although this approach is hardly applicable to all problems, it fits some quite well; serious production systems have been successfully created in this way.

Usage in development projects. Shell programming has been used for years to support programming projects. As one example, one of the authors manages a team of software people, systems engineers, and psychologists in producing a management decision support system now being deployed in the Bell System. Shell procedures are used extensively to help manage the project. The result is a cohesive, integrated, heavily automated environment, used not only by programmers, but by everyone involved in the project, including managers, planners, and end users. Simple procedures exist to control the product, regenerate it from the source code, and even deliver it electronically to remote machines, all without much manual effort. Other procedures are used to avoid repetitive coding by converting skeleton programs or lists of data items into complete C or PL/I programs. Some programs are transformed to become documents, and vice versa, so that people tend to view the system as an integrated data base of project information and procedures, in which the line between software and documentation is quite fuzzy.

Heavily integrated programming environments are proposed often, but few are actually built; fewer still are successful. An irony of the environment described above is that its outward appearance is of a comprehensive, integrated system. In fact, it is made up of just the opposite—small shell procedures and Unix commands.

This project also illustrates the use of shell procedures in the delivered product. A hybrid mainframe-minicomputer product, it contains about 15K lines of PL/I, 10K lines of C, 30K lines of documentation, and 16K lines of shell. Shell procedures provide most of the user-visible functions. This project's success has often depended on the ability to write something quickly using the shell, obtain user feedback, and adapt it rapidly to fit real needs discovered in the field. In numerous cases, the first version of a program was written quickly and then discarded just as quickly—not because it was slow, but because requirements changed as soon as end users saw the results of those requirements.

Unix and modern programming methodologies

Even though at its birth a system may be clean and easy to use, the natural increase of entropy tends to cause it to grow ugly and unpleasant. Like any other system, Unix is vulnerable to this process, although so far it has aged gracefully. Fortunately, its creators have always favored taste, restraint, and minimality of construct.[5,14,2] They have maintained a steady pressure to reduce the number of system calls, subroutines, and commands by judicious generalization or by combination of similar constructs.

In some environments, every new construct is hailed as an advance—an expression of the philosophy that more is always better. Unix developers view additional constructs with suspicion, while greeting with pleasure proof that several existing constructs can be combined and simplified as a result of some new insight. Anything new must prove that it truly deserves a niche in the scheme of things, and it must then hold its place against competition. In the long run, any given niche really has room for but one occupant, so people continually attempt to identify distinct niches and fill them with the fittest competitors.

The capacities of human beings to comprehend, document, and maintain computer software have limits, which must be respected; therefore, redundant and overlapping software must be avoided. It is especially important to maintain simplicity in constructs that are central to everyone's use. No feature is truly free; each has costs as well as benefits, which must be weighed carefully before the feature is included in central programs like the Unix kernel, the C compiler, the shell, or the text editor.

System evolution. The current Unix system has evolved through a process resembling Darwinian selection. In the first stage of the cycle (mutation), the "standard" version of the system is used by many people, who inevitably customize it for their local needs. They usually lobby to have their favorite features included in the next standard version. In the second stage (selection and cross-breeding), the next standard version is created and often includes features from the strongest mutants. Meanwhile, the weaker mutants die out, since people tire of supporting unnecessary differences. Although the Unix system has grown more complex in this process, features usually have been validated in more than one project before inclusion in the standard version.

Programming methodologies can also be selected by first encouraging experimentation, then eliminating the least competitive approaches. No one can afford to swallow the entire deluge of available methodologies, for each addition seems to produce fewer results than its predecessor. Thus, one should pick and choose with care. Although there is no panacea for programming ills, Unix usage seems to solve many common problems without bother or new methodologies. There are two reasons for

The Unix system supports many approaches in such a natural and pervasive way that people apply them without great effort, often without awareness of the published literature.

this. First, the Unix system supports many approaches in such a natural and pervasive way that people apply them without great effort, often without awareness of the published literature. Second, other approaches are made unnecessary by using the Unix system in the first place. Some examples follow.

Structured coding is taken for granted, since modern control-flow constructs are provided by C, the shell, and most other language processors used on Unix systems. The code that people see, adapt, and imitate is usually well structured. People learn to code well in the same way that they learn to speak their native language well, by imitation and immediate feedback.

Formal walk-throughs are used only occasionally on Unix systems, because people often look at each other's code, comment on it in person and through interuser communication facilities, and take pieces of it for their own use. The ideas of programming teams and egoless programming fit into the Unix environment well, since they encourage sharing rather than isolation. Although some programs have always been "owned" by one or two people, many others have been passed around so much that it is difficult to tell exactly who wrote them. Programming groups of widely varying personalities exist happily in the Unix environment; neither chief programmers nor truly egoless ones are common.

Design techniques such as data flow diagrams, pseudocode, and structure charts are seldom necessary, especially in light of the ease of writing a few short shell procedures to provide the code and documentation for the highest levels of control. For example, Yourdon and Constantine[15] say ". . . it is definitely true that many Unix designers (the authors included) do nothing more than program the bubbles in a data flow graph, without the intermediate step of converting it into a structure chart." Certain aspects of the *Jackson Design Methodology,*[16] such as program inversion and resolution of structure clashes, seem unnecessary in a system that provides pipes, allows the use of small programs, and eliminates logical and physical records.

The idea of a development support library is justifiably popular.[17] The Unix system performs the services required of such a library efficiently and conveniently, especially when compared to packages grafted onto existing batch systems. Since the latter were originally built for different purposes, their communication and file systems are often not oriented to interactive work.

Baker[17] has observed that the exact role of the program librarian in an interactive development environment "remained to be determined." In the presence of a Unix system the role seems to be minimal, especially in the original sense of providing control and eliminating drudgery. Programs, documents, test data, and test output are stored in the Unix file system and protected either by the usual file access mechanism or by more elaborate software, such as the Source Code Control System.[18,19] Much of the drudgery found in other systems is simply bypassed by Unix; it has always been fit for human beings to use. Tools have been built to automate many common programming tasks,[20] and project control procedures are easily written as shell procedures. Many of our programming groups have experimented with the librarian concept and concluded that, given a decent environment, there is little need for a program librarian.

None of this should necessarily be taken as a criticism of these techniques, which can be useful in some situations. We simply prefer to minimize the number of techniques we must use to get a job done, and we observe that Unix service is the last one we would give up.

Attributes of programming environments

Programming environments resemble programming languages. Most people use only a few, prefer either their first or their current one above all others, and argue the merits of one versus another. And yet, there are few truly objective metrics for comparison. In this section, we suggest some important attributes by which to classify programming environments and analyze the design tradeoffs found in them. The Unix system is evaluated in these terms.

Group size, organization, and sociology. Programming environments usually reflect the size, organization, and sociology of the groups that create them. At one extreme is the loosely coupled set of individuals, each with his own computer or isolated virtual machine. Examples include personal computers, the Xerox Alto/Ethernet architecture, and perhaps IBM's VM370/CMS. At the other extreme, some systems are built specifically to handle the problems of large programming teams. ICL's CADES,[21] which supported a 200-person project, is an example of a successful system of this type.

Unix systems typically lie between the two extremes. Through 1974, Unix best supported a single, cooperative, tightly coupled group of people on each machine. By 1975, PWB/Unix began offering better support for larger groups and multiple groups per machine. Over the years, the mechanisms needed to support this variety have evolved and been included in the standard system. Although some large projects (of more than 200 people) are successfully supported on networks of Unix machines, most people who work together prefer to use the same machine so they can easily share procedures and data bases.

System adaptability. This attribute measures the ease of adapting a system to change its capabilities. Systems at one extreme have many capabilities, but every capability is fixed, unchangeable by the user. At the other extreme, the system provides few features and is easy to change; or else provides a good toolkit, but requires that tools be combined to do the job. The first extreme offers a system optimized for some job; the second offers flexibility, but at the cost of modifying a system or assembling some tools. The first is often easier for the beginner; experts tend to favor the second.

The Unix system favors a minimum of built-in constructs and maximum ease of adaptability. Ease of change is sometimes a disadvantage. Because it is easy to change the operating system, variants proliferate—often for no good reason. A second disadvantage is that new users can be overpowered by the toolkit provided. They know there is a way to do a job, but there are so many tools that it is difficult to find the right one.

Level of expertise. Systems often aim at different levels of expertise and at different mixtures of expertise level. Early Unix systems were used and supported by expert programmers. Later versions have tended to add more facilities for less-experienced people. Often, a programming group contains a single "guru," who customizes the general environment to support the specific needs of people less skilled in the use of Unix tools.

Life cycle. Assume that a software product's life cycle consists of requirements analysis, design, code, test, and deployment, with maintenance considered to be repeated cycles of the previous steps. Different programming environments support these phases unequally. For example, PSL/PSA[22] and similar systems emphasize requirements analysis. Many systems emphasize coding or testing aids. Unix systems seldom support ambitious packages for requirements analysis and design, but do support these activities in general ways, e.g., by providing good text-processing tools and file organizations that are convenient for managing data bases of requirements and design information. PWB/Unix was somewhat unusual for its time, in emphasizing deployment, support, and maintenance tools and in specifically attacking time-consuming programming drudgery.

Integration of facilities. At one extreme—found all too often—system features are independent to the extent that they are difficult to use together, even when it is necessary to do so. At the other extreme, a few systems offer tight integration of as many facilities as possible, so that the system displays a comprehensive, uniform interface to the user. Good examples are Interlisp[23,24] and Smalltalk.[25]

The Unix toolkit approach lies somewhere in the middle. Most tools are specifically written to be modular and independent, even if the result is a set of tools rather than a single large command that could be used in their place. This approach has some disadvantages that strongly integrated systems do not. Sometimes commands use unnecessarily different calling conventions, simply because they are contributed by different people. In cases where commands are often used together or interact in peculiar ways, users would prefer to have more integrated combinations. For example, there are circumstances in which the one-way flow of information in a pipe is just not adequate. This is particularly evident in complicated text-processing applications, in which preprocessors need better communication with TROFF than a pipe can provide.

Specialization. Systems differ radically in their views of specialization, not only for attributes mentioned above, but for others such as language, methodology, and target computer. At one extreme is a system that supports one choice very well and others not at all. As examples, one finds "single-language" systems and program development systems that not only support but enforce a single choice of methodology. At the other extreme lie those systems that would support all choices in a fairly equal fashion.

Unix systems lie in the middle. Although they support a number of languages, C and the shell are usually supported significantly better than anything else. Unix systems are fairly impartial with regard to methodology, and they support the production of code for many different target execution environments. The best-supported target is the Unix system itself. Next best are those non-Unix systems, including most microprocessors, that are dominated by their supporting Unix systems and therefore often use interfaces built for development convenience. Next are those machines for which Unix-based support software has been written, but which dominate the attached Unix systems so that interfaces may well be less convenient than desired. Examples include special-purpose computers (switching machines) and large, non-Unix mainframes that enforce their own interfaces. Finally, some targets are not supported at all, and thus require the writing of new interfaces.

Failure and success. Winners and losers can seldom be distinguished solely by studying the published literature, which contains proposals for systems never built, glowing accounts that outshine the actual user manuals, and descriptions of systems that work but have not spread beyond their original environments. People seldom write retrospectives on failures; when written, they often go unpublished. In this section, we propose a success scale for programming environments and place Unix on that scale.

Extreme failure is represented by a system that is never built at all, or by a paper system whose ideas are never

A truly successful system's use spreads far beyond its original inventors. It is used enthusiastically by many people and quickly occupies niches from which it is difficult to dislodge.

taken up by any successful system.

Medium failure is represented by a system that is built but soon abandoned, even by its immediate inventors. If a system is used only by its inventors and would vanish upon their departure, it probably falls into this category.

Many programming environments enjoy minor success. They are used by people other than the original inventors and are recognized further afield, but obtain only a small slice of the potential market.

Extreme success has several attributes. A truly successful system's use spreads far beyond its original inventors. It is used enthusiastically by many people and quickly occupies niches from which it is difficult to dislodge. People build new systems using its ideas or even parts of its software. Finally, people assume the system is so well known that they mention it without citation.

Given its exponential growth rate and widespread impact, it is not surprising that the Unix system rates quite high on this scale. One can trace the spread of many software tools from the Unix system through a book[26] into various other environments.[27-29] Much of the current research on secure systems is Unix-based,[30-32] many industry and government groups have chosen Unix as the basis for or inclusion in their own programming environment systems,[33-38] and numerous vendors offer Unix-related products.

Reasons for success. To be successful, any programming environment must score well on at least some technical attributes. However, technical quality is not enough for success—many high-quality systems have fallen by the wayside. At least part of the Unix system's success must come from its adaptability and nonspecialization, which allow it to thrive in many different niches.

Success or failure often depends on nontechnical factors, whose importance often goes unrecognized by those who evaluate systems on purely technical terms. For example, a system is seldom widely successful if it is based on "orphan" hardware, i.e., hardware that is out of production or produced only in small numbers. Unix popularity was certainly not harmed by the initial choice of hardware, the DEC PDP-11.

Another impediment to success is a high rating on what we call the "gulp factor." Suppose that an organization must swallow a new system in a giant gulp because it requires a great deal of time or money, demands massive changes in programming techniques, or requires abandoning much existing code. Such a system is much harder to spread than is a system like Unix, which is relatively cheap, can be integrated slowly into the existing environment, and can help with the maintenance of old code. Small Unix systems have often opened doors for later, larger machines—the original Programmer's Workbench/Unix installation started as a single PDP-11/45 in 1973 and currently contains over 20 PDP-11/70s and VAX-11/780s. Some of that expansion depended on the ability to pick up existing code (Cobol, for example) and assist in its maintenance. The decreasing ratio of new development to maintenance indicates that any programming environment faces an uphill battle if it insists that all existing code be discarded and that all work be done from the very beginning of the project life cycle.

Finally, it is a nontechnical plus for a system to be fun to use. This factor alone can make up for many other deficiencies.

Conclusions

We have found the Unix environment to be an especially productive one. This is largely because it presents a clean and systematic interface to programs that run on it, it has a wealth of small, well-designed programs that can serve as building blocks in larger processes, and it provides mechanisms by which these programs can be quickly and effectively combined. The programmable command language itself is the single most important such program, for it provides the means by which most other programs cooperate.

These facilities are used for a wide range of applications. Design, coding, and debugging are all made easier by the use of combinations of existing, small, reliable components instead of the construction of new, large, unreliable ones. Finally, the Unix system goes a long way toward solving people's programming problems without requiring a host of additional tools and methodologies.

The Unix system is exceptionally successful—we hope this article helps to explain why and how this success came about. ■

Acknowledgments

Ken Thompson and Dennis Ritchie are the creators of the Unix system; without their insight and good taste, none of this would be possible. We are also indebted to the many people who have built on the Unix base.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 365-375.
2. D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," *Bell System Technical J.,* Vol. 57, No. 6, Oct. 1978, pp. 1947-1969.
3. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *Bell System Technical J.,* Vol. 57, No. 6, Oct. 1978, pp. 1991-2019.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1978.
5. K. Thompson, "The UNIX Command Language," in *Structured Programming—Infotech State of the Art Report,* Infotech International Ltd., Berkshire, England, Mar. 1975, pp. 375-384.
6. S. R. Bourne, "An Introduction to the UNIX Shell," *Bell System Technical J.,* Vol. 57, No. 6, Oct. 1978, pp. 2797-2822.
7. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell System Technical J.,* Vol. 57, No. 6, Oct. 1978, pp. 2021-2048.

8. Richard Miller, "UNIX—A Portable Operating System?" *Operating Systems Rev.*, Vol. 12, No. 3, July 1978, pp. 32-37.

9. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," *Proc. 2nd Int'l Conf. Software Eng.*, Oct. 1976, pp. 164-168.

10. E. L. Ivie, "The Programmer's Workbench—A Machine for Software Development," *Comm. ACM*, Vol. 20, No. 10, Oct. 1977, pp. 746-753.

11. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell System Technical J.*, Vol. 57, No. 6, Oct. 1978, pp. 2177-2200.

12. J. R. Mashey, "Using a Command Language as a High-Level Programming Language," *Proc. 2nd Int'l Conf. Software Eng.*, Oct. 1976, pp. 169-176.

13. T. A. Dolotta and J. R. Mashey, "Using a Command Language as the Primary Programming Tool," in *Command Language Directions: Proc. 79 IFIP Working Conf. Command Languages*, D. Beech, ed., North-Holland, Amsterdam, The Netherlands, 1980.

14. D. M. Ritchie, "The Evolution of the UNIX Time-Sharing System," *Proc. Symp. Language Design and Programming Methodology*, Sidney, Australia, 1979.

15. E. Yourdon and L. L. Constantine, *Structured Design*, Yourdon Press, New York, 1978, p. 317.

16. M. A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.

17. F. T. Baker, "Structured Programming in a Production Programming Environment," *Proc. Int'l Conf. Reliable Software*, 1975, pp. 172-185.

18. M. J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 4, Dec. 1975, pp. 364-370.

19. A. L. Glasser, "The Evolution of a Source Code Control System," *SICSOFT*, Vol. 3, No. 5, Nov. 1978, pp. 121-125.

20. S. I. Feldman, "MAKE—A Program for Maintaining Computer Programs," *UNIX Programmer's Manual*, Vol. 9, Apr. 1979, pp. 255-265.

21. D. J. Pearson, "The Use and Abuse of a Software Engineering System," *AFIPS Conf. Proc.*, 1979 NCC, pp. 1029-1035.

22. D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 42-48.

23. W. Teitelman, *INTERLISP Reference Manual*, Xerox Corp. Palo Alto Research Center, Palo Alto, Calif., Dec. 1978.

24. W. Teitelman, "A Display Oriented Programmer's Assistant," CSL 77-3, Xerox Corp. Palo Alto Research Center, Palo Alto, Calif., Mar. 1977.

25. A. Kay and A. Goldberg, "Personal Dynamic Media," *Computer*, Mar. 1977, pp. 31-41.

26. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass., 1976.

27. D. E. Hall, D. K. Scherrer, and J. S. Sventek, "A Virtual Operating System," *Comm. ACM*, Vol. 23, No. 9, Sept. 1980, pp. 495-502.

28. C. R. Snow, "The Software Tools Project," *Software—Practice & Experience*, Vol. 8, No. 5, Sept.-Oct. 1978.

29. P. H. Enslow, Jr., *Portability of Large Cobol Programs: The Cobol Programmer's Workbench*, Georgia Institute of Technology, Atlanta, Ga., Sept. 1979.

30. J. P. L. Woodward, "Applications for Multilevel Secure Operating Systems," *AFIPS Conf. Proc.*, 1979 NCC, June 1979, pp. 319-328.

31. G. J. Popek et al., "UCLA Secure Unix," *AFIPS Conf. Proc.*, 1979 NCC, June 1979, pp. 355-364.

32. E. J. McCauley and P. J. Drongowski, "KSOS—The Design of a Secure Operating System," *AFIPS Conf. Proc.*, 1979 NCC, June 1979, pp. 345-353.

33. E. J. McCauley, G. L. Barksdale, and J. Holden, "Software Development Using a Development Support Machine," *ADA Environment Workshop*, DoD High Order Language Working Group, Nov. 1979, pp. 1-9.

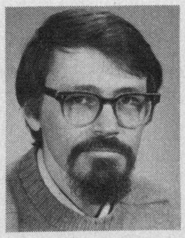
34. M. Risenberg, "Software Costs Can Be Tamed, Developers Told," *Computerworld*, Jan. 29, 1980, pp. 1-8.

35. J. E. Stockenberg and D. Taffs, "Software Test Bed Support Under PWB/UNIX," *ADA Environment Workshop*, DoD High Order Language Working Group, Nov. 1979, pp. 10-26.

36. R. A. Allshouse, D. T. McClellan, G. E. Prine, and C. P. Rolla, "CSDP as an ADA Environment," *ADA Environment Workshop*, DoD High Order Language Working Group, Nov. 1979, pp. 113-125.

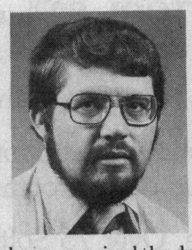
37. P. Wegner, "The ADA Language and Environment," *Proc. Electro/80*, Western Periodicals Co., North Hollywood, Calif., May 1980.

38. R. A. Robinson and E. A. Krzysiak, "An Integrated Support Software Network Using NSW Technology," *AFIPS Conf. Proc.*, 1980 NCC, May 1980, pp. 671-676.



Brian W. Kernighan is a member of the technical staff in the Computing Principles Research Department, Bell Laboratories, Murray Hill, New Jersey. His current research activities are in document preparation software, programming languages, and programming methodology. He is the coauthor (with P. J. Plauger) of *The Elements of Programming Style, Software Tools,* and *Software Tools in Pascal,* and (with D. M. Ritchie) of *The C Programming Language.*

He received a BS in engineering physics from the University of Toronto in 1964, and an MA and PhD in electrical engineering from Princeton University in 1969.



John R. Mashey is a supervisor of a software development group at Bell Laboratories, Whippany, New Jersey. His interests include programming methodology, command languages, text processing, and interactions of computers, people, and personnel structure. Since joining Bell Laboratories in 1973, he has worked on the Programmer's Workbench version of Unix (PWB/Unix) and, since mid-1978, has supervised the design and development of a new applications system, used to reduce the costs of telephone facilities maintenance.

He is an affiliate member of the IEEE, a member of ACM, and has been an ACM National Lecturer for several years. He received a BS in mathematics in 1968, an MS in computer science in 1969, and a PhD in computer science in 1974, all from the Pennsylvania State University.