UNOS 1447

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 1-15 (1979)

The UNIX™ Programming Environment

BRIAN W. KERNIGHAN

Bell Laboratories, Murray Hill, New Jersey 07974, U. S. A.

JOHN R. MASHEY

Bell Laboratories, Whippany, New Jersey 07981, U. S. A.

SUMMARY

The UNIX* operating system provides an especially congenial programming environment, in which it is not only possible, but actually natural, to write programs quickly and well.

Several characteristics of the UNIX system contribute to this desirable state of affairs. Files have no type or internal structure, so data produced by one program can be used by another without impediment. The basic system interface for input and output provides homogeneous treatment of files, I/O devices and programs, so programs need not care where their data comes from or goes to. The command interpreter makes it convenient to connect programs, by arranging for all data communication.

Complex procedures are created not by writing large programs from scratch, but by interconnecting relatively small components. These programs are small and concentrate on single functions, and therefore are easy to build, understand, describe, and maintain. They form a high level toolkit whose existence causes programmers to view their work as the use and creation of tools, a viewpoint that encourages growth in place of reinvention.

Tools interact in a limited number of ways, but can be used in many different combinations. Thus, an addition to the toolkit tends to improve the programming power of the user faster than it increases the complexity of interconnection and maintenance. Finally, tools are connected at a very high level by a powerful command language interpreter. The error-prone and expensive process of program writing can often be avoided in favor of program-using.

In this paper we will present a variety of examples to illustrate this methodology, focusing on those aspects of the system and supporting software which make it possible.

KEY WORDS Operating systems Programmer productivity

INTRODUCTION

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

There is more than a grain of truth in this remark, even though it probably wasn't meant the way it sounds. In particular, 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, awkward, and present major obstacles to simply getting a job done.

Things needn't be that way. For at least five years, we have used the UNIX operating system[1]

* UNIX is a trademark of Bell Laboratories.

0038-0644/79/0109-0001$01.00

Received 31 July 1978

which in many ways is just the opposite — 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 an attic by two people in a year, and has been available only as an unsupported package, the benefits it provides are so compelling that currently there are nearly 1000 UNIX systems scattered around the world. At Bell Laboratories, UNIX systems provide more time-sharing ports than all other systems combined; they are accessed by thousands of people, many on a daily basis.

This paper is not meant to be an extended UNIX advertisement. It is our intent to describe what appears to be a new way of computing, emphasizing those things that are unique, that are particularly well done, or that are especially good for programmer productivity. We will also discuss aspects that have changed our view of the actual programming process, and attempt to draw some lessons that may be valuable to future implementors of operating systems.

A disclaimer — neither of us was involved with the development of the UNIX system, although we have contributed applications software. We thus describe the system from the user's viewpoint, as derived from our own experiences and those of the large community of users with whom we have been involved.

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 may 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.

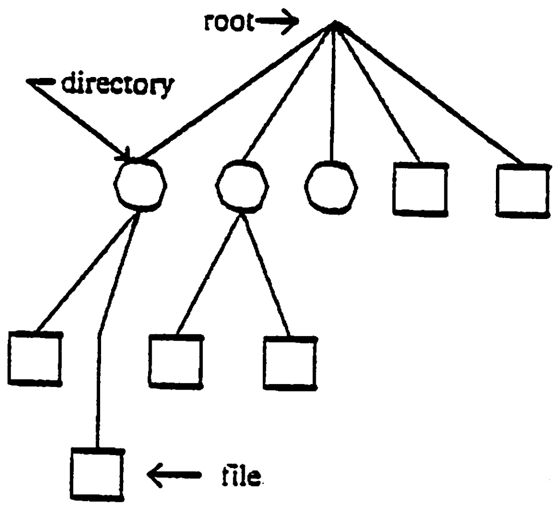

Figure 1. File system hierarchy

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 is necessary since directories contain information about the physical structure of the file system.)

Since directories are files, they may 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 attributes that 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 nor 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 is. 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 no file types for different kinds of data nor are there 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 a major contribution 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. 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** may have any positive value.

Both **read** and **write** return the number of bytes actually transferred. This may be less than **n**, for example when reading a file whose size is not a multiple of **n** bytes. A return of zero on reading marks 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, but it converts the haphazard suboptimizations of individual user programs into global optimization across the entire set of active programs. Disk performance is especially improved by this approach.

I/O is normally sequential — each command continues where the preceding one left off. This default may be changed by the call seek:

```
seek(fd, position, relative_to)
```

This requests that the pointer for the next read or write be set to **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. It is written in the C programming language.[2,3]

```
fin = open(name1, READ);
fout = open(name2, WRITE);
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 when the command 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, 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. 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, rather than 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 may be devices:

```
copy /device/tape /device/printer
```

copies the magnetic tape onto the printer, and

```
copy /device/telephone /device/terminal
```

reads data from the telephone onto a user's terminal. The program **copy** is in all cases identical. the four lines we wrote above. The **copy** program need not concern itself with any special characteristics of files, magnetic tape or paper tape, 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 a lot simpler to have but one **copy** program than to have 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, inter-user 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 unwritable by others.

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" 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

```
Fri May 5 22:31:30 EDT 1978
```

The "program name" is in fact 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. (In fact, 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 system used for computing research, for example, has about 25000 files for about 40 active users: the average file size is about 10000 bytes, but the median is very 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 probably impossible for a longer list. The shell, however, provides an equivalent 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 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; 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
```

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 replacing:

```
who >>userlist
```

appends the new information to the end of **userlist**.

The text editor is called **ed**:

```
ed <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 that assumption. On other systems, it is possible to perform the redirection, but difficult. But it is not enough that something 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 a representative sample, among the more useful. We will use them as illustrations of other points in later sections of the paper.

diff oldfile newfile
print differences between two files

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 **goto**'s.

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 multi-column list of the file names on the on-line printer. We use the multi-column 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 of the programs involved 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 to capture the outputs, when all that is being done is to take 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 non-programming users.

Although in principle the pipe notation could be merely a shorthand for the longer form with temporaries, there are 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, but 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 or a file or a pipe. Commands can be written in the simplest possible way, yet used in a variety of contexts without prearrangement. (Bear in mind that 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 normally 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; and a number of postprocessors for **troff** that produce output on various media other than the typesetter. Placing all of these facilities into one typesetting language and program would have created an absolutely unworkable monster. As it is, however, each piece is sufficiently independent that it can 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, merely because the intermediate states are clearly visible and can be materialized in files at any time. (As an aside, this paper has been printed directly from camera-ready copy produced by **refer** and **troff**.)

Since programs can interact, novel interactions spring up. As an instance, consider the 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 spare 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, nor tell you that **joe** is logged on). Instead they concentrate on doing one thing well; they are designed to interact with other programs; and the system provides an easy and elegant way to do the connection. The interconnections are limited not by preconceptions built into the system, but by people's imaginations.

In this environment, people begin to search consciously 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', etc. Rather than writing a special sort or modifying the existing one, he wrote the trivial program **rev** that reverses each line of its input. Then

```
rev <dictionary | sort | rev >rhymingdict
```

does the job. (Notice 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** in this paper: 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 225 programs — most of

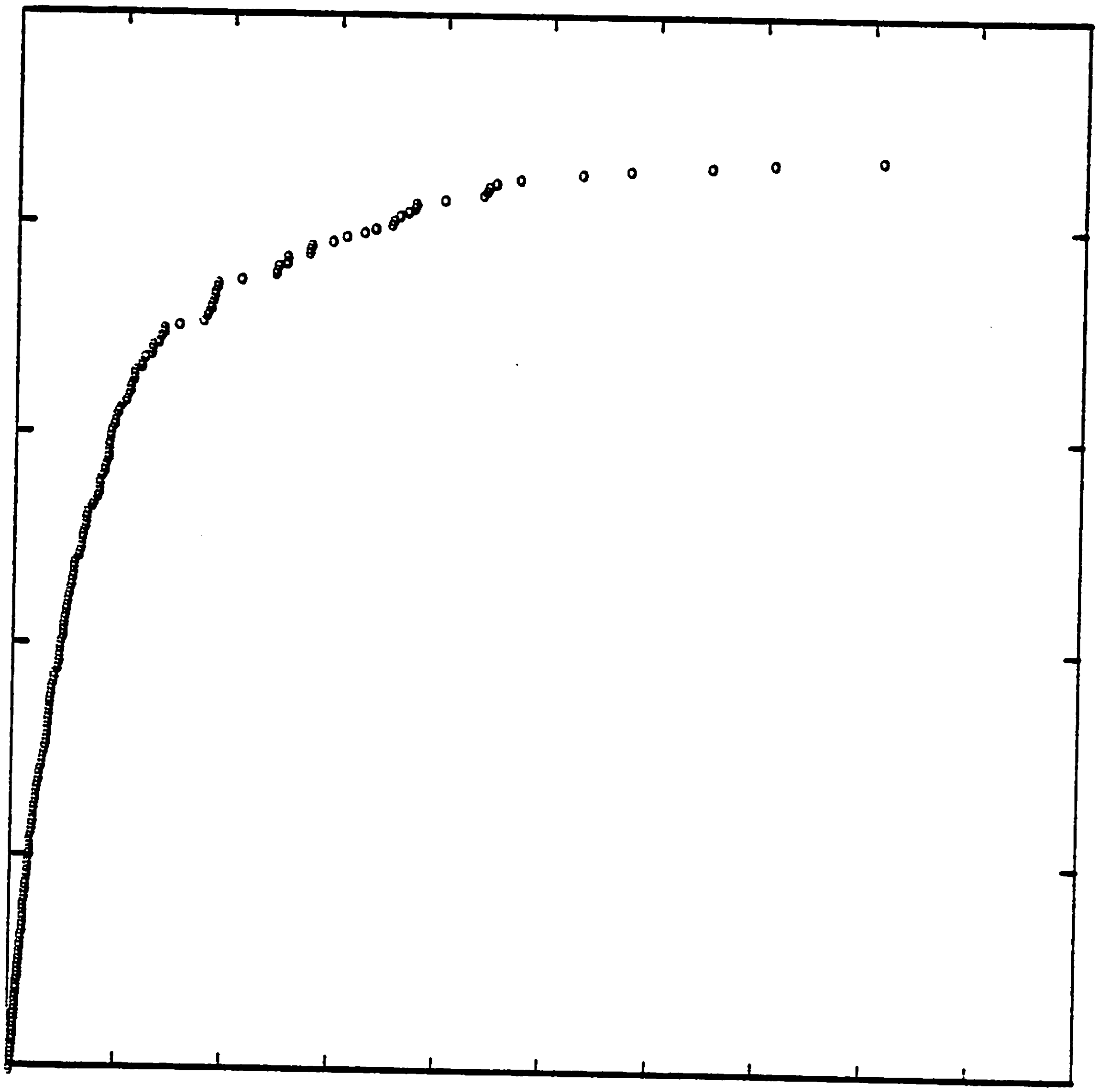

Figure 2. Program Sizes on UNIX. x-axis (0, 5000)

the commonly used commands, but excluding compilers — was counted with wc. The programs have been sorted into increasing order of number of source lines. The *x* axis is the number of lines; the *y* axis is simply the ordinal number of the program.

The median program here is slightly over 240 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 with a hundred 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 at the same time 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*, that is, that will compile and run without change on other machines. It is now routine in our environment that programs developed for the UNIX system are exported to other systems unchanged. There is obviously a considerable gain in productivity in not having to rewrite the same program over again for each new machine.

Since the operating system itself and all of its software is written in C, this leads to the interesting possibility of a portable operating system, one that can be moved with little effort from one piece of hardware to another. At the moment, the UNIX system itself has been moved from the PDP-11 to the Interdata 8/32 and the VAX 11/780. From the user's standpoint, these systems are indistinguishable, save that not all of the standard software is necessarily available on the newer systems. More details may be found in the paper by Johnson and Ritchie.[4] An independent experiment by Miller[5] describes transporting UNIX to the Interdata 7/32.

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.[6] This has several implications. It can readily evolve to meet changing requirements. It 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, and so its input too may be redirected with <. Thus if a set of commands are placed in a file, they may be executed just as if they had been typed, by the command

```
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, placed in a file instead. Thus a "catalogued procedure" facility is a natural by-product of the standard I/O mechanism, not a special case.

This is such a useful capability that several steps have been taken to make it even more valuable. The first is to add a limited macro capability: if there are arguments on the command line that invokes 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, and it 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 tiny example, consider the shell program **tel**, which uses **grep** to search an ordinary text file **/usr/lib/tel** for telephone numbers or names or whatever. 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 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 towards writing complicated processes in the shell instead of in C. Throughout, however, it has remained true that the shell is just an ordinary user program: its input is ordinary text, and a user can not, by running a program, determine whether or not it is a shell process.

Although the shell resembles a typical procedural language, it has certain rather different qualities. Most important, in certain ways it is a *very* high level language, and as such it 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 — people can create new commands on the spur of the moment. It can be adapted to meet performance requirements without disturbing its user interface. The elements of its language are generally quite independent — 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 for even a beginner to write unmodular shell procedures — modularity is inherent in the language, without effort or careful preplanning.

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 nine that make up the original Programmer's Workbench (PWB/UNIX) installation.[7,8] The system serves about 350 users, who own a total of 39,000 files and 2850 directories. The mean file size is about 3700 bytes; the mean number of files per directory is 14. A majority of the people using this system are working on programs to be run on IBM S/370 computers; some are working on software to run on a UNIX system; everyone uses the system for documentation and project management activities. Project sizes range from 1 person to about 50.

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. These shell procedures 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 1. In fact, 11.7% of all procedures consisted of but a single line. About 45-50% of the procedures contained some conditional logic; of these, about half (or 20-25% of the total) included loops, primarily to perform the same operation over each file in an argument list. Programming usage has increased, as can be observed by comparison with a previous survey.[9]

Several conclusions can be drawn. First, people make significant use of shell procedures to customize the general environment to their particular needs, even 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 may still be quickly customized for local needs. Second, programming goals are accomplished by writing shell procedures rather than compiled programs. Examples include small database management packages, procedures to generate complex Job Control Language sequences, and project management procedures for configuration control, system regeneration, project scheduling, data dictionary management, and inter-user communication. Third, as people become accustomed to this methodology, its use increases with time. Such heavy use of shell programming is not found on the original research-oriented UNIX machine, but is common practice at many development-oriented installations. Shell programming has been used for years to support programming projects. We are now starting to see development projects in which the delivered code consists mainly of shell procedures.

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**, **wc**, and so on 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 editor **ed** that we mentioned earlier.

Second, if a program is necessary, the initial version can often be written as a shell procedure instead of 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 mentioned before.

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 any experience may well lead to a first-class solution of the wrong problem.) A program may require a period of rapid and drastic evolution before it converges toward stability. 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, a procedure may evolve to a more-or-less stable state, i.e., it does what it ought to. At this point, it can be left alone if it is fast enough for its intended uses, which are by now well known. But if it is too slow, it can be entirely rewritten in C, or at least some small, crucial section recoded, with the existing version providing a proven functional specification. Deferring efficiency considerations until the design and the usage patterns have stabilized and until a need has been shown usually avoids 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 gets written, its requirements are fairly well defined. Or an existing tool may 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 in place of unreliable or expensive programming. The effort required to produce both reliability and efficiency is reserved for code that really requires these attributes, and is applied efficiently, because accurate requirements are known by the time the code is written.

Even though at its birth, a system may be clean and easy to use, the natural process of entropy may cause it to grow ugly and unpleasant. Like any other system, UNIX is vulnerable to this process, although so far it has escaped relatively unscathed. Fortunately, its creators have always favored taste, restraint, and minimality of construct.[6, 10] There is a steady pressure to reduce the number of system calls, subroutines, and commands by judicious generalization or combination of similar constructs.

In some environments, every new construct is hailed as an advance, following the philosophy that more is always better. UNIX developers tend to view additional constructs with suspicion, while greeting with pleasure proof that several existing constructs can be combined and simplified, presumably because some insight has been achieved. Anything new must prove that it truly deserves a "niche" in the scheme of things, and it must fight to keep its place against competition. In the long run, any given niche really has but room for 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 finite limits, which must be respected by avoiding the creation of redundant and overlapping software. It is especially important to maintain the simplicity of constructs that are truly central to everyone's use. No "feature" is truly free; each has costs as well as benefits, and they must be weighed carefully, especially when considered for inclusion in central programs like the UNIX kernel, the C compiler, the shell, or the text editor.

A similar point of view can be applied to the use of programming methodologies. No one can afford to swallow the entire deluge of available methodologies, for each addition seems to buy less results than its predecessor. Thus, one should pick and choose methodologies with care. Although nothing is a panacea for all programming ills, UNIX usage seems to solve many common problems without fuss and bother, and without requiring herculean efforts devoted to learning and applying sets of new methodologies. We have observed two distinct kinds of effects. First, the UNIX system supports many approaches in such a natural and pervasive way that people apply them without great effort, and often without becoming aware of the published literature. Second, other approaches are simply made unnecessary by using the UNIX system in the first place. Some specific 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, i.e., by imitation and immediate feedback.

Formal *walk-throughs* are used only occasionally on UNIX systems, because people often look at other people's code, comment on it in person and through inter-user communication facilities, and take pieces of it for their own use. The ideas of *programming teams* and *egoless programming* fit well into the UNIX environment, 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.

Design techniques such as *data flow diagrams*, *pseudocode*, and *structure charts* are seldom viewed as necessary, especially when compared with the ease of writing a few short shell procedures to provide the code and documentation for the highest levels of control.

Certain aspects of the *Jackson Design Methodology*[11] 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.[12] The UNIX system performs the services required of such a library, and performs them efficiently and conveniently, especially when compared to packages grafted onto existing batch systems. Since the latter were originally built for different purposes, they often lack communication and file systems oriented to interactive work.

Baker[12] 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.[13] 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; 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 that we must use to get a job done, and we observe that UNIX service is the *last* one that we would give up.

CONCLUSIONS

We have found the UNIX environment to be an especially productive one. In large part this is because the system itself presents a clean and systematic interface to programs that run on it; there is a wealth of small, well-designed programs that may be used as building blocks in larger processes; and the system provides mechanisms by which these programs may 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.

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

ACKNOWLEDGEMENTS

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. Assoc. Comp. Mach.*, 17, 7, 365-375 (1974).
2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, 'UNIX Time-Sharing System: The C Programming Language,' *Bell Sys. Tech. J.*, 57, 6, 1991-2019 (1978).
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
4. S. C. Johnson and D. M. Ritchie, 'UNIX Time-Sharing System: Portability of C Programs and the UNIX System,' *Bell Sys. Tech. J.*, 57, 6, 2021-2048 (1978).
5. Richard Miller, 'UNIX -- A Portable Operating System?,' *Operating Systems Review*, 12, 3, 32-37 (1978).
6. K. Thompson, 'The UNIX Command Language,' in *Structured Programming—Infotech State of the Art Report*, Infotech International Ltd., 375-384 (1975).
7. T. A. Dolotta and J. R. Mashey, 'An Introduction to the Programmer's Workbench,' *Proc. 2nd Int. Conf. on Software Engineering* 164-168 (1976).
8. E. L. Ivie, 'The Programmer's Workbench—A Machine for Software Development,' *Comm. Assoc. Comp. Mach.*, 20, 10, 746-753 (1977).
9. J. R. Mashey, 'Using a Command Language as a High-Level Programming Language,' *Proc. 2nd Int. Conf. on Software Engineering* 169-176 (1976).
10. D. M. Ritchie, 'UNIX Time-Sharing System: A Retrospective,' *Bell Sys. Tech. J.*, 57, 6, 1947-1969 (1978).
11. M. A. Jackson, *Principles of Program Design*, Academic Press, 1975.
12. F. T. Baker, 'Structured Programming in a Production Programming Environment,' *Proc. Int. Conf. on Reliable Software* 21, 172-185 (1975).
13. M. J. Rochkind, 'The Source Code Control System,' *IEEE Trans. on Software Engineering*, SE-1, 4, 364-370 (1975).