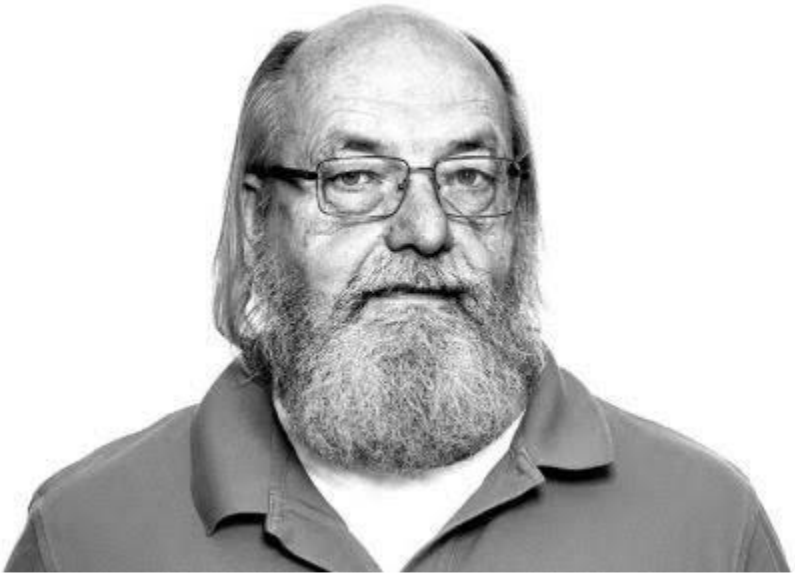
PROGRAMMER AT WORK: Ken Thompson Publications



Kenneth Lane Thompson (born February 4, 1943) is an American pioneer of computer science. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating system. Since 2006, Thompson has worked at Google, where he co invented the Go programming language.

Ken Thompson

Born	Kenneth Lane Thompson February 4, 1943 (age 76) New Orleans, Louisiana, U.S.
Nationality	American
Alma mater	University of California, Berkeley (B.S., 1965; M.S., 1966)
Known for	Unix B (programming language) Belle (chess machine) UTF-8

	Plan 9 from Bell Labs Inferno (operating system) Endgame tablebase Go
Awards	IEEE Emanuel R. Piore Award (1982) Turing Award (1983) Member of the National Academy of Sciences (1985) IEEE Richard W. Hamming Medal (1990) Computer Pioneer Award (1994) National Medal of Technology (1998) Tsutomu Kanai Award (1999) Harold Pender Award (2003) Japan Prize (2011)
	Scientific career
Fields	Computer science
Institutions	Bell Labs Entrisphere, Inc Google

- **Password Security: A Case History**
- **Unix and Beyond: An Interview with Ken Thompson**
- **A New C Compiler**
- **Reflections on Trusting Trust**
- **Plan 9 from Bell Labs**
- **UNIX Implementation**
- **The UNIX Time-Sharing System**
- **Regular Expression Search Algorithm**
- **Hello World**

Operating Systems
R. Stockton Gaines Editor

Password Security: A Case History

Robert Morris and Ken Thompson
Bell Laboratories

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

Key Words and Phrases: operating systems, passwords, computer security
CR Categories: 2.41, 4.35

Introduction

Password security on the UNIX (a trademark of Bell Laboratories) time-sharing system [3] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

Authors' present address: R. Morris and K. Thompson, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e., prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so-called "super-user" password on the UNIX system, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual authorized access and accidental disclosure.

Prologue

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60s at a time when one of

the authors (Morris) happened to be using the system. A system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

The First Scheme

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [4, p. 91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [1] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

Attacks on the First Approach

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search*. Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g., all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alpha-numeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	44 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.	112 days	500 days
6	107 hrs.	760 hrs.	2.2 yrs.	29 yrs.	174 yrs.

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typically about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

—The dictionary with the words spelled backwards.
—A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
—The above with initial upper-case letters.
—All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
—Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time,

15 were a single ASCII character;
72 were strings of two ASCII characters;
464 were strings of three ASCII characters;
477 were strings of four alphamerics;
706 were five letters, all upper-case or all lower-case;
605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831 or 86 percent of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

An Anecdote

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudorandom number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time. In this case, no harm was done, as the bad guy happened to be a friendly user.

Improvements to the First Approach

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4,096 (2^{12}). The reason for this is that there are 4,096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be very high.

5. A Subtle Point

To log in successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in an invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

Conclusions

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is set up to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. For example, an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

Received August 1978; revised August 1979

References

1. Hagelin, B. Ciphering Machine (M–209), U.S. Patent No. 2,089,603, Aug. 10, 1937.
2. Proposed federal information processing data encryption standard. Federal Register (40FR12134), March 17, 1975.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
4. Wilkes, M.V. *Time-Sharing Computer Systems.* American Elsevier, New York, 1968.

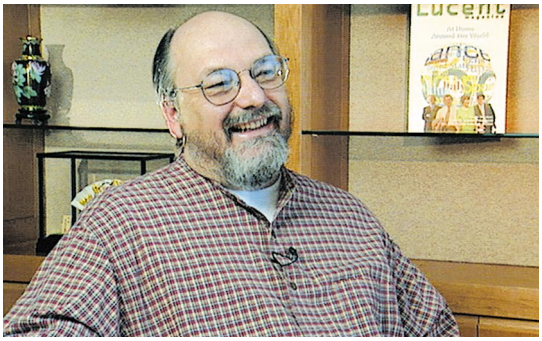
Unix and Beyond: An Interview with Ken Thompson

Computer recently visited Ken Thompson at Lucent's Bell Labs to learn about Thompson's early work on Unix and his more recent research in distributed computing.

Daniel Cooke
Texas Tech University

Joseph Urban
Arizona State University

Scott Hamilton
Computer



Ken Thompson needs no introduction: the co-creator of the Unix operating system as well as the Plan 9 and Inferno distributed operating systems; creator, along with Joseph Condon, of Belle, a world champion chess computer; 1998 US National Medal of Technology winner, along with Dennis Ritchie, for their role in developing the Unix system and C.

On the occasion of the presentation of the Computer Society's and Hitachi's inaugural Tsutomu Kanai Award for distributed computing, *Computer* visited recipient Ken Thompson at Lucent's Bell Labs. We were interested in learning about Thompson's early work on Unix and his more recent work in distributed computing. We were especially interested in learning about the creative process within Bell Labs and his sense of where computer science is heading.

CREATIVITY AND SOFTWARE DEVELOPMENT

Computer: Your nominators and endorsers for the Kanai Award consistently characterized your work as simple yet powerful. How do you discover such powerful abstractions?

Thompson: It is the way I think. I am a very bottom-up thinker. If you give me the right kind of Tinker Toys, I can imagine the building. I can sit there and see primitives and recognize their power to build structures a half mile high, if only I had just one more to make it functionally complete. I can see those kinds of things.

The converse is true, too, I think. I *can't*—from the building—imagine the Tinker Toys. When I see a top-down description of a system or language that has infinite libraries described by layers and layers, all I just see is a morass. I can't get a feel for it. I can't understand how the pieces fit; I can't understand something presented to me that's very complex. Maybe I do what I do because if I built anything more complicated, I couldn't understand it. I really must break it down into little pieces.

Computer: In your group you probably have both the bottom-up thinker and the top-down thinker. How do you interact with both?

Kanai Award

For more information on the Kanai Award, see "With Two New Awards, We Honor Unix, RISC Innovators," pp. 11-13.

0018-9162/99/$10.00 © 1999 IEEE

Thompson: I think there's room for both, but it makes for some interesting conversations, where two people think they are talking to each other but they're not. They just miss, like two ships in the night, except that they are using words, and the words mean different things to both sides. I don't know how to answer that, really. It takes both; it takes all kinds.

Occasionally—maybe once every five years—I will read a paper and I'll say, "Boy, this person just doesn't think like normal people. This person thinks at an orthogonal angle." When I see people like that, my impulse is to try to meet them, read their work, hire them. It's always good to take an orthogonal view of something. It develops ideas.

I think that computer science in its middle age has become incestuous: People are trained by people who think one way. As a result, these so-called orthogonal thinkers are becoming rarer and rarer. Of course, many of their ideas have become mainstream—like message passing, which I thought was something interesting when I first saw it. But occasionally you still see some very strange stuff.

Software development paradigms

Computer: What makes Plan 9 and the Inferno network operating system very striking is the consistent and aggressive use of a small number of abstractions. It seems clear that there's a coherent vision and team assembled here working on these projects. Could you give us further insight into how the process works?

Thompson: The aggressive use of a small number of abstractions is, I think, the direct result of a very small number of people who interact closely during the implementation. It's not a committee where everyone is trying to introduce their favorite thing. Essentially, if you have a technical argument or question, you have to sway two or three other people who are very savvy. They know what is going on, and you can't put anything over on them.

As for the process, it's hard to describe. It's chaotic, but somehow something comes out of it. There is a structure that comes out of it. I am a member of the Computing Sciences Research Center, which consists of a bunch of individuals—no teams, no leaders. It's the old Bell Labs model of research; these people just interact every day.

At different times you have nothing to do. You've stopped working for some reason—you finished a project or got tired of it—and you sit around and look for something to do. You latch on to somebody else, almost like water molecules interacting.

You get together and say, "I have an idea for a language," and somebody gets interested. Somebody else asks how we put networking in it. Well, so-and-so has a model for networking, and somebody else comes in. So you have these teams that rarely get above five or six, and usually hover around two or three. They each bring in whatever they did previously.

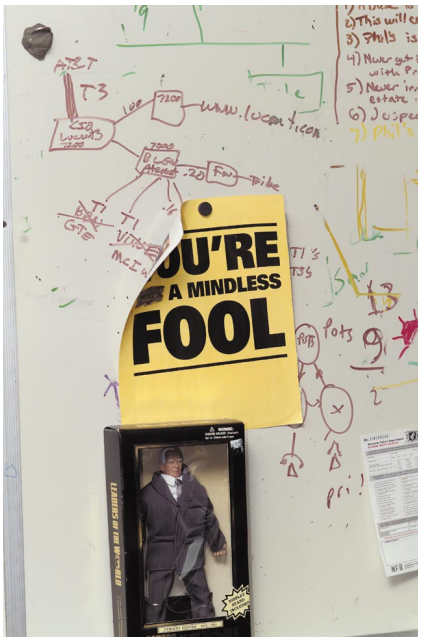
So that's the way it works. There are no projects per se in the Computing Sciences Research Center. There are projects near it of various sorts that will draw on our research as a resource. But they have to deal with our style. If people get stuck, they come to us but usually don't want to deal with the management style—which means none—that comes along with it.

Computer: You mentioned technical arguments and how you build your case. How are technical arguments resolved?

Thompson: When you know something is systemically wrong despite all the parts being correct, you say there has to be something better. You argue back and forth. You may sway or not sway, but mostly what you do is come up with an alternative. Try it. Many of the arguments end up that way.

You say, "I am right, the hell with you." And, of course the person who has been "to helled with" wants to prove his point, and so he goes off and does it. That's ultimately the way you prove a point. So that is the way most of the arguments are done—simply by trying them.

I don't think there are many people up in research who have strong ideas about things that they haven't really had experience with. They won't argue about the theory of something that's never been done. Instead, they'll say, "Let's try this." Also, there's not that much ego up there either, so if it's a failure you come back and say, "Do you have another idea? That one didn't work." I have certainly generated as many bad ideas as I have good ones.



Photos of the "Unix Room" at Bell Labs courtesy of Lucent Technologies.

Computer: What advice do you have for developers who are out there now to improve their designs so that they could be viewed as producing simple yet powerful systems?

Thompson: That is very hard; that is a very difficult question. There are very few people in my position who can really do a design and implement it. Most people are a smaller peg in a big organization where the design is done, or they do the design but can't implement it, or they don't understand the entire system. They are just part of a design team. There are very few people I could give advice to.

It's hard to give advice in a product kind of world

when what I do, I guess, is some form of computer Darwinism: Try it, and if it doesn't work throw it out and do it again. You just can't do that in a product-development environment.

Plus I am not sure there are real principles involved as opposed to serendipity: You happened to require this as a function before someone else saw the need for it. The way you happen upon what you think about is just very lucky. My advice to you is just be lucky. Go out there and buy low and sell high, and everything will be fine.

UNIX

Computer: In an earlier interview you were asked what you might do differently if you had to do Unix over again, and you said that you would add an "e" to the `creat` *system call. Seriously, in hindsight, can you give us an assessment of the problems you overcame, the elegant solutions, and the things you would have done differently.*

Thompson: I think the major good idea in Unix was its clean and simple interface: open, close, read, and write. This enabled the implementation of the shell as well as Unix's portability. In earlier systems, I/O had different entry points, but with Unix you could abstract them away: You open a file, and if the file happens to be a tape, you could write to it. Pipes allowed tools and filters that could accommodate classical monster programs like `sort`.

Probably the glaring error in Unix was that it underevaluated the concept of remoteness. The open-close-read-write interface should have been encapsulated together as something for remoteness; something that brought a group of interfaces together as a single thing—a remote file system as opposed to a local file system.

Unix lacked that concept; there was just one group of open-close-read-write interfaces. It was a glaring omission and was the reason that some of the awful things came into Unix like `ptrace` and some of the system calls. Every time I looked at later versions of Unix there were 15 new system calls, which tells you something's wrong. I just didn't see it at the time. This was fixed in a fairly nice way in Plan 9.

Computer: Going back a little bit further, what were the good and not so good aspects of Multics that were the major drivers in the Unix design rationale?

Thompson: The one thing I stole was the hierarchical file system because it was a really good idea—the difference being that Multics was a virtual memory system, and these "files" weren't files but naming conventions for segments. After you walk one of these hierarchical name spaces, which were tacked onto the side and weren't really part of the system, you touch it and it would be part of your address space and then you use machine instructions to store the data in that segment. I just plain lifted this.

By the same token, Multics was a virtual memory system with page faults, and it didn't differentiate between data and programs. You'd jump to a segment as it was faulted in, whether it was faulted in as data or instructions. There were no files to read or write—nothing you could remote—which I thought was a bad idea. This huge virtual memory space was the unifying concept behind Multics—and it had to be tried in an era when everyone was looking for the grand unification theory of programming—but I thought it was a big mistake.

I wanted to separate data from programs, because data and instructions are very different. When you're reading a file, you're almost always certain that the data will be read sequentially, and you're not surprised when you fault a and read $a + 1$. Moreover, it's much harder to excise instructions from caches than to excise data. So I added the `exec` system call that says "invoke this thing as a program," whereas in Multics you would fault in an instruction and jump to it.

Development history

Computer: What about the development history of Unix?

Thompson: The early versions were essentially me experimenting with some Multics concepts on a PDP-7 after that project disbanded, which is about as small a team as you can imagine. I then picked up a couple of users, Doug McIlroy and Dennis Ritchie, who were interested in languages. Their criticism, which was very expert and very harsh, led to a couple of rewrites in PDP-7 assembly.

At one point, I took BCPL from Martin Richards at MIT and converted it into what I thought was a fairly straight translation, but it turned out to be a different language so I called it B, and then Dennis took it and added types and called it C.

We bought a PDP-11—one of the very first—and I rewrote Unix in PDP-11 assembly and got it running. That was exported to several internal Bell telephone applications, to gather trouble reports and monitor various things like rerouted cables. Those applications, independent of what we were doing, started political pressure to get support for the operating system; they demanded service. So Bell Labs started the Unix Support Group, whose purpose was to serve as the interface to us, to take our modifications and interface them with the applications in the field, which demanded a more stable environment. They didn't like surprises.

This grew over time into the commercial version from AT&T and the more autonomous version from USL.

Independently, we went on and tried to rewrite Unix in this higher level language that was evolving simultaneously. It's hard to say who was pushing whom—whether Unix was pushing C or C was pushing Unix. These rewrites failed twice in the space of six months, I believe, because of problems with the language. There would be a major change in the language and we'd rewrite Unix.

The third rewrite—I took the OS proper, the kernel, and Dennis took the block I/O, the disk—was successful; it turned into version 5 in the labs and version 6 that got out to universities. Then there was a version 7 that was mostly a repartitioning of the system in preparation for Steve Johnson and Dennis Ritchie making the first port to an Interdata 832. Unknown to us, there was a similar port going on in Australia.

Around version 6, ARPA [Advanced Research Projects Agency] adopted it as the standard operating system for the Arpanet community. Berkeley was contracted to maintain and distribute the system. Their major contributions were to adapt the University of Illinois TCP/IP stack and to add virtual memory to Bell Lab's port to the VAX.

There's a nice history of Unix written by Dennis that's available on his home page [*ed.*—"The Evolution of the Unix Time-Sharing System," http://cm.belllabs.com/cm/cs/who/dmr/hist.html].

Computer: What accounted for the success of Unix, ultimately?

Thompson: I mostly view it as serendipitous. It was a massive change in the way people used computers, from mainframes to minis; we crossed a monetary threshold where computers became cheaper. People used them in smaller groups, and it was the beginning of the demise of the monster comp center, where the bureaucracy hidden behind the guise of a multimillion-dollar machine would dictate the way computing ran. People rejected the idea of accepting the OS from the manufacturer and these machines would never talk to anything but the manufacturer's machine.

I view the fact that we were caught up in that—where we were glommed onto as the only solution to maintaining open computing—as the main driving force for the revolution in the way computers were used at the time.

There were other, smaller things. Unix was a very small, understandable OS, so people could change it at their will. It would run itself—you could type "go" and in a few minutes it would recompile itself. You had total control over the whole system. So it was very beneficial to a lot of people, especially at universities, because it was very hard to teach computing from an IBM end-user point of view. Unix was small, and you could go through it line by line and understand exactly how it worked. That was the origin of the so-called Unix culture.

Computer: In a sense, Linux is following in this tradition. Any thoughts on this phenomenon?

Thompson: I view Linux as something that's not Microsoft—a backlash against Microsoft, no more and no less. I don't think it will be very successful in the long run. I've looked at the source, and there are pieces that are good and pieces that are not. A whole bunch of random people have contributed to this source, and the quality varies drastically.

My experience and some of my friends' experience is that Linux is quite unreliable. Microsoft is *really* unreliable but Linux is *worse*. In a non-PC environment, it just won't hold up. If you're using it on a single box, that's one thing. But if you want to use Linux in firewalls, gateways, embedded systems, and so on, it has a long way to go.

DISTRIBUTED COMPUTING: NETWORK OPERATING SYSTEMS AND LANGUAGES

Computer: How does your work on Plan 9 and Inferno derive from your earlier work on Unix? What are some of the new ideas arising out of this work that could and should apply to distributed operating systems in general?

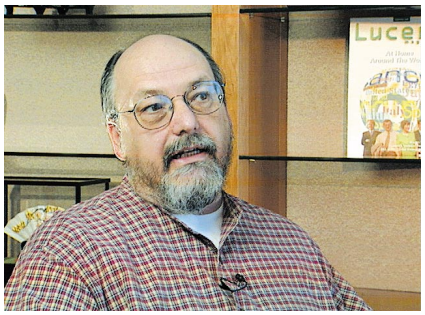
Thompson: Saying these ideas haven't been applied before is tough because, if you look closely, everything is reinvented, nothing's new. There are good ideas and bad ideas in Unix. You can't escape your history. What you think today is not much different from what you thought yesterday. And, by induction, it is not that different from what you thought twenty years ago.

In Plan 9 and Inferno, the key ideas are the protocol for communicating between components and the simplification and extension of particular concepts. In Plan 9, the key abstraction is the file system—anything you can read and write and select by names in a hierarchy—and the protocol exports that abstraction to remote channels to enable distribution. Inferno works similarly, but it has a layer of language interaction above it through the Limbo language interface—which is like Java, but cleaner, I think.

Limbo

Computer: How would you characterize Limbo as a language?

Thompson: First, I have to say that the language itself is almost exclusively the work of Sean Dorward, and in my talking about it I don't want to imply I had much to do with it.



I think it's a good language. In a pragmatic sense, it's a simplification of the larger languages like C++ and Java. The inheritance rules are much simpler, it's easier to use, and the restrictions there for simplicity don't seem to impair its functionality.

In C++ and Java I experience a certain amount of angst when you ask how to do this and they say, "Well, you do it like this or you could do it like that." There are obviously too many features if you can do something that many ways—and they are more or less equivalent. I think there are smaller concepts that fit better in Inferno.

Computer: We know that Plan 9 was done in C. It would almost seem that the group needed Limbo to develop Inferno. Do we need new types of languages to build distributed systems?

Thompson: The language, I think, doesn't matter per se. The language's actual size and features are almost separate issues from the distribution of the language. It shouldn't be too large or too small; it should be some nice language that you can live with. The idea, though, is that it is dynamically loadable so that you can replace little modules. And through some other mechanisms like encryption you can validate those modules, and when they are loaded you have some confidence that it's the module you wanted and that someone hasn't spoofed you.

There are certain features you must have—some form of object orientation, for example. You could replace Limbo with Java—I wouldn't want to—and not change Inferno's basic principles other than the way it meets system requirements. Sean decided the whole system had to have a garbage-collected language at a much higher level in that it's not separate interacting processes maintaining their own addresses, with some being garbage-collected and some not.

The language and the system are all garbage-collected together. Whatever protection mechanisms you have for the language apply all the way down through the system. For example, if you open a file, you don't have to close it. If you stop using it, just return from the function and it will be garbage-collected and the file will be closed. So the system and the language are part of the same ball of wax.

In addition, the language implementation—and again I don't want to take any credit—doesn't have big mark-and-sweep type garbage collection. It has reference counting: If you open something and then return, it's gone by reference count. Thus, you don't have high and low watermarks because 99 percent of the garbage goes away as soon as it is dereferenced. If you store a null in a pointer, it chases the pointer and all that stuff goes.

If you make cycles, there is a background distributed mark-and-sweep algorithm that just colors the next step a little bit at a time. It doesn't have to be very aggressive because there is not much garbage around in most applications: People really don't leave dangling loop structures that require this kind of algorithm. So you can devote just one percent of your time in the background looking for garbage without these monster mark-and-sweep things.

So, again, it's pragmatic. It's not the theoretical top-of-the-line garbage collection paper. It's just a way of doing it that seems to be very, very effective.

CURRENT WORK

Computer: What are you working on now?

Thompson: A few of us in research were tapped by a newly formed development organization within Lucent to work on a product called the PathStar Access Server. It's essentially a central office switch and router for IP phone and data services. It's strictly IP-based. You pick up the phone, dial it, and make conference calls.

I think packet switching will replace circuit switching in the phone system and will invert the hierarchy. Whereas data is currently carried in the leftover space of a circuit-switched network, eventually the backbone will be a packet-switched network with the phone implemented under it. You don't have to go out on a limb to say this—probably 90 percent of the people believe that now. But this project is "put up or shut up." We are actually inverting the phone system to run across a pretty classical packet-switched router.

In this kind of application what you need to pay attention to is maintenance and configuration, which is where Inferno comes in. All of the configuration code is Inferno and Limbo. You have to pay attention to quality of ser-

vice so that you can raise the loading above minimal and still get real-time voice, in this particular case.

There were some fun parts: The actual call processing, which is typically done by a huge finite state machine, was fun to do. We did it by making a finite-state-machine-generation language. The object of the language is a finite state machine, but the source is not. The actual phone conversation or feature is a group of interacting finite state machines, almost like processes. And, of course, they have to be distributed because you make calls to other phones.

Computer: So this language generates the finite state machines. Did you create the language to allow for experimentation to come up with different finite state machines?

Thompson: Well, at first we thought it was simple: You just write a finite state machine for this phone system. And at first it *was* simple. You just say, "Well if you're here do this, and if you're there do that, and just manually lay out these finite state machines." And that works just beautifully for the very first implementation, which is just picking up a phone, dialing a number, calling another phone, picking that phone up, conversing, and hanging up. You can just picture those states laying out.

But when you get to some of the simple features—three-way calls, for example—what happens when caller ID or call waiting comes in on a three-way call? The classical phone just says busy because it can't handle more than three phones.

So you build a model, which was initially a finite-state-machine model, and then you slowly add the features you need until the model breaks. It breaks pretty quickly, so you build a second one until it breaks, and so on. You just do it by exhaustion. So that's how the FSM-generation language came about; it wasn't "let's sit down and do everything at once." I think that's probably the way computer languages were built.

Interestingly, this work was extended further by Gerard Holzmann, someone in our area who has been into state verification—running exhaustive studies finding error states in the finite state system. He was just delighted with this little FSM-generation language because now he could build his models, and he inverted it. He took it the way it is, which is to build finite state machines, but he also took it to build drivers. So he has my model, which runs the phones on the inside, but then he needed telephones to drive his model. So he can now build another finite state machine to model the telephone and do not only the synthesis but the analysis.

Jukebox music collection

Computer: You're also collecting music?

Thompson: It's kind of a personal/research hobby/project. Let me explain it from an external point of view. Basically, I'm just collecting music. I'm getting lists from various sources—top 10s, top 50s—and I try to collect the music.

Right now, my list has around 35,000 songs, of which I've collected around 20,000. I compress the songs with a Bell-Labs-invented algorithm called PAC [Perceptual Audio Coding] and store them on a jukebox storage system. I started this before MP3 was heard of on the network. PAC is vastly superior to MP3.

My collection is not generally available because of the legal aspects. I went to legal and told them I was collecting a lot of music, but I don't think they realized what I meant by "a lot." Anyway, they said that in the case of research there's something similar to fair use and that they'd back me, but wouldn't go to jail for me. So I can't release it generally. But it's pretty impressive. It's split-screen like a Web browser; you can walk down lists, years, or weeks.

Computer. It's a personal hobby.

Thompson: It's hard to differentiate since, if you haven't noticed, almost everything I've done is personal interest. Almost everything I've done has been supported and I'm allowed to do it, but it's always been on the edge of what's acceptable for computer science at the time. Even Unix was right on the edge of what was acceptable at Bell Labs at the time. That's almost been my history.

COMPUTER SCIENCE AND THE FUTURE

Computer: You've been there through Multics, Unix, Inferno, and so on. Any thoughts about where computer science is going or should be going?

Thompson: Well, I had to give advice to my son, and my advice to him—to the next generation—was to get into biology.

Computer science is coming into its middle age. It's turning into a commodity. People don't know about Carnot cycles for refrigerators, yet they buy refrigerators. It's happening in computing, too. Who knows about compilers? They buy computers to play games and balance their checkbooks. So my advice to my child was—I am unfortunately talking to *Computer* magazine—to go into biology, not classic biology but gene therapy and things like that.

I think that computing is a finite field and it's reaching its apex and we will be on a wane after this. I am

sorry to say that, but that's the way I feel. You look at any aspect of computer science—what's being taught today, PhD theses, publications, any metric you can think of and compare it to history—and you realize that aspects of computer science are becoming more specialized.

Computer: Which aspects?

Thompson: Operating systems, in particular, have to carry so much baggage. Today, if you're going to do something that will have any impact, you have to compete with Microsoft, and to do that you have to carry the weight of all the browsers, Word, Office, and everything else. Even if you write a better operating system, nobody who actually uses computers today knows what an operating system interface is; their interface is the browser or Office.

You can have the best and most beautiful interface in the world and the most extensible operating system that ports to anything, and then you have to port on top of it a thousand staff-years' worth of applications that you can't obtain the source for. You have two choices: Go to Microsoft and ask for the source to Office to port to your operating system and they'll laugh at you; or get a user's manual and reengineer the code and they'll sue you anyway. Basically, it'll never happen because the entry fee is too high.

Anything new will have to come along with the type of revolution that came along with Unix. Nothing was going to topple IBM until something came along that made them irrelevant. I'm sure they have the mainframe market locked up, but that's just irrelevant. And the same thing with Microsoft: Until something comes along that makes them irrelevant, the entry fee is too difficult and they won't be displaced.

Computer: So you're not precluding the possibility of a paradigm shift.

Thompson: Absolutely not. Anybody who says there's no more innovation in the world is doomed to be among the last 400 people who have stated this since the birth of Christ.

Computer: You're still having fun?

Thompson: Yes, there are still a lot of fun programs to write.

WHAT I DID ON MY WINTER VACATION

Computer: We can't let you go without asking why on earth you traveled to Russia to fly a Mig-29?

Thompson: How often does the Soviet Union collapse? It would be just a shame if you couldn't do something you have always wanted to do as a result. They are selling rides in what was once the top fighter. A mere two years earlier you would only get hints about its existence in Jane's books. Now you can get in, use the laser sights, and go straight up at 600 miles per hour. Who wouldn't do that? When things like that come along, I'll take them. They're fun. ❖

Acknowledgments

We thank Patrick Regan of Lucent Technologies for arranging this interview and Brigitta Hanggi for providing the photos. Photos of the "Unix Room" in the Computing Sciences Research Center are courtesy of Lucent Technologies.

Daniel Cooke is chair of the Computer Science Department at Texas Tech University.

Joseph Urban is a professor in the Computer Science and Engineering Department at Arizona State University.

Scott Hamilton is senior acquisitions editor for Computer.

Contact the authors at {d.cooke, j.urban, s.hamilton}@computer.org.

A New C Compiler

Ken Thompson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes yet another series of C compilers. These compilers were developed over the last several years and are now in use on Plan 9. These compilers are experimental in nature and were developed to try out new ideas. Some of the ideas were good and some not so good.

1. Introduction

Most C compilers consist of a multitude of passes with numerous interfaces. A typical C compiler has the following passes – pre-processing, lexical analysis and parsing, code generation, optional assembly optimisation, assembly (which itself is usually multiple passes), and loading. [Joh79a]

If one profiles what is going on in this whole process, it becomes clear that I/O dominates. Of the cpu cycles expended, most go into conversion to and from intermediate file formats. Even with these many passes, the code generated is mostly line-at-a-time and not very efficient. With these conventional compilers as benchmarks, it seemed easy to make a new compiler that could execute much faster and still produce better code.

The first three compilers built were for the National 32000, Western 32100, and an internal computer called a Crisp. These compilers have drifted into disuse. Currently there are active compilers for the Motorola 68020 and MIPS 2000/3000 computers. [Mot85, Kan88]

2. Structure

The compiler is a single program that produces an object file. Combined in the compiler are the traditional roles of pre-processor, compiler, code generator, local optimiser, and first half of the assembler. The object files are binary forms of assembly language, similar to what might be passed between the first and second passes of an assembler.

Object files and libraries are combined and loaded by a second program to produce the executable binary. The loader combines the roles of second half of the assembler, global optimiser, and loader. There is a third small program that serves as an assembler. It takes an assembler-like input and performs a simple translation into the object format.

3. The Language

The compiler implements ANSI C with some restrictions and extensions. [Ker88] If this had been a product-oriented project rather than a research vehicle, the compiler would have implemented exact ANSI C. Several of the poorer features were left out. Also, several extensions were added to help in the implementation of Plan 9. [Pik90] There are many more departures from the standard, particularly in the libraries, that are beyond the scope of this paper.

3.1. Register, volatile, const

The keywords `register`, `volatile`, and `const`, are recognised syntactically but are semantically ignored. `Volatile` seems to have no meaning, so it is hard to tell if ignoring it is a departure from the standard. `Const` only confuses library interfaces with the hope of catching some rare errors.

`Register` is a holdover from the past. Registers should be assigned over the individual lives of a variable, not on the whole variable name. By allocating registers over the life of a variable, rather than by pre-allocating registers at declaration, it is usually possible to get the effect of about twice as many registers. The compiler is also in a much better position to judge the allocation of a register variable than the programmer. It is extremely hard for a programmer to place register variables wisely. When one does, the code is usually optimised to a particular compiler or computer. The portability of the performance of a program with register declarations is poor.

There is a semantic feature of a declared register variable in ANSI C – it is illegal to take its address. This compiler does not catch this ‘‘mistake.’’ It would be easy to carry a flag in the symbol table to rectify this, but that seems fussy.

3.2. The pre-processor

The C pre-processor is probably the biggest departure from the ANSI standard. Most of differences are protests about common usage. Some of the difference is due to the generally poor specification of the existing pre-processors prior to the ANSI report.

This compiler does not support `#if`, though it does handle `#ifdef`. In practice, `#if` is almost always followed by a variable like ‘‘pdp11.’’ What it means is that the programmer has buried some old code that will no longer compile. Another common usage is to write ‘‘portable’’ code by expanding all possibilities in a jumble of left-justified chicken scratches.

As an alternate, the compiler will compile very efficient normal `if` statements with constant expressions. This is usually enough to rewrite old `#if`-laden code.

If all else fails, the compiler can be run with any of the existing pre-processors that are still maintained as separate passes.

3.3. Unnamed substructures

The most important and most heavily used of the extensions is the declaration of an unnamed substructure or subunion. For example:

```
struct      lock
{
      int    locked;
} *lock;

struct      node
{
      int   type;
      union
      {
            double dval;
            float  fval;
            long   lval;
      };
      struct lock;
} *node;
```

This is a declaration with an unnamed substructure, `lock`, and an unnamed subunion. This shows the two major usages of this feature. The first allows references to elements of the subunit to be accessed as if they were in the outer structure. Thus `node->dval` and `node->locked` are legitimate references. In C, the name of a union is almost always a non-entity that is mechanically declared and used with no purpose.

The second usage is poor man's classes. When a pointer to the outer structure is used in a context that is only legal for an unnamed substructure, the compiler promotes the type. This happens in assignment statements and in argument passing where prototypes have been declared. Thus, continuing with the example,

```
lock = node;
```

would assign a pointer to the unnamed lock in the node to the variable `lock`. Another example,

```
extern void lock(struct lock*);
func(...)
{
	...
	lock(node);
	...
}
```

will pass a pointer to the lock substructure.

It would be nice to add casts to the implicit conversions to unnamed substructures, but this would conflict with existing C practice. The problem comes about from the almost ambiguous dual meaning of the cast operator. One usage is conversion; for example `(double)5` is a conversion, but `(struct lock*)node` is a PL/1 ''unspec.''

3.4. Structure displays

A structure cast followed by a list of expressions in braces is an expression with the type of the structure and elements assigned from the corresponding list. Structures are now almost first-class citizens of the language. It is common to see code like this:

```
r = (Rectangle){point1, (Point){x,y+2}}.
```

3.5. Initialisation indexes

In initialisers of arrays, one may place a constant expression in square brackets before an initialiser. This causes the next initialiser to go in that indicated element. This feature comes from the expanded use of `enum` declarations. Example:

```
enum  errors
{
	Etoobig,
	Ealarm,
	Egreg,
};
char* errstrings[] =
{
	[Etoobig]   "Arg list too long",
	[Ealarm]    "Alarm call",
	[Egreg]     "Panic out of mbufs",
};
```

This example also shows a micro-extension – it is legal to place a comma on the last `enum` in a list. (Wow! What were they thinking?)

3.6. External register

The declaration `extern register` will dedicate a register to a variable on a global basis. It can only be used under special circumstances. External register variables must be identically declared in all modules and libraries. The declaration is not for efficiency, although it is efficient, but rather it represents a unique storage class that would be hard to get any other way. On a shared-memory multi-processor, an external register is one-per-machine and neither one-per-procedure (automatic) or one-per-system (external). It is

used for two variables in the Plan 9 kernel, *u* and *m*. *U* is a pointer to the structure representing the currently running process and *m* is a pointer to the per-machine data structure.

3.7. Goto case, goto default

The last extension has not been used, so is probably not a good idea. In a switch it is legal to say `goto case 5` or `goto default` with the obvious meaning.

4. Object module conventions

The overall conventions of the runtime environment are very important to runtime efficiency. In this section, several of these conventions are discussed.

4.1. Register saving

In most compilers, the called program saves the exposed registers. This compiler has the caller save the registers. There are arguments both ways. With caller-saves, the leaf subroutines can use all the registers and never save them. If you spend a lot of time at the leaves, this seems preferable. In called-saves, the saving of the registers is done in the single point of entry and return. If you are interested in space, this seems preferable. In both, there is a degree of uncertainty about what registers need to be saved. The convincing argument is that with caller-saves, the decision to registerise a variable can include the cost of saving the register across calls.

Perhaps the best method, especially on computers with many registers, would to have both caller-saved registers and called-saved registers.

In the Plan 9 operating system, calls to the kernel look like normal subroutine calls. As such the caller has saved the registers and the system entry does not have to. This makes system call considerably faster. Since this is a potential security hole, and can lead to non-determinisms, the system may eventually save the registers on entry, or more likely clear the registers on return.

4.2. Calling convention

Rule: ‘‘It is a mistake to use the manufacturer’s special call instruction.’’ The relationship between the (virtual) frame pointer and the stack pointer is known by the compiler. It is just extra work to mark this known point with a real register. If the stack grows towards lower addresses, then there is no need for an argument pointer. It is also at a known offset from the stack pointer. If the convention is that the caller saves the registers, then the entry point saves no registers. There is therefore no advantage to a special call instruction.

On the National 32100 computer programs compiled with the simple ‘‘jsr’’ instruction would run in about half the time of programs compiled with the ‘‘call’’ instruction.

4.3. Floating stack pointer

On computers like the VAX and the 68020, there is a short, fast addressing mode to push and pop the top of stack. In a sequence of subroutine calls within a basic block, arguments may be pushed and popped many times. Pushing arguments is, to some extent, a useful activity, but popping is just overhead. If the arguments of the first call are left on the stack for the second call, a single pop of both sets of arguments (usually an ‘‘add’’ instruction) will suffice for both calls. This optimisation is worth several percent in both space and runtime of object modules.

The only penalty comes in debugging, when the distance between the stack pointer and the frame pointer must be communicated as a program counter-dependent variable rather than a single variable for an entire subroutine.

4.4. Functions returning structures

Structures longer than one word are awkward to implement since they do not fit in registers and must be passed around in memory. Functions that return structures are particularly clumsy. These compilers pass the return address of a structure as the first argument of a function that has a structure return value. Thus

```
x = f(...)
```

is rewritten as

```
f(&x, ...).
```

This saves a copy and makes the compilation much less clumsy. A disadvantage is that if you call this function without an assignment, a dummy location must be invented. An earlier version of the compiler passed a null pointer in such cases, but was changed to pass a dummy argument after measuring some running programs.

There is also a danger of calling a function that returns a structure without declaring it as such. Before ANSI C function prototypes, this would probably be enough consideration to find some other way of returning structures. These compilers have an option that complains every time that a subroutine is compiled that has not been fully specified by a prototype, which catches this and many other errors. This is now the default and is highly recommended for all ANSI C compilers.

5. Implementation

The compiler is divided internally into four machine-independent passes, four machine-dependent passes, and an output pass. The next nine sections describe each pass in order.

5.1. Parsing

The first pass is a YACC-based parser. [Joh79b] All code is put into a parse tree and collected, without interpretation, for the body of a function. The later passes then walk this tree.

The input stream of the parser is a pushdown list of input activations. The preprocessor expansions of `#define` and `#include` are implemented as pushdowns. Thus there is no separate pass for preprocessing.

Even though it is just one pass of many, the parsing take 50% of the execution time of the whole compiler. Most of this (75%) is due to the inefficiencies of YACC. The remaining 25% of the parse time is due to the low level character handling. The flexibility of YACC was very important in the initial writing of the compiler, but it would probably be worth the effort to write a custom recursive descent parser.

5.2. Typing

The next pass distributes typing information to every node of the tree. Implicit operations on the tree are added, such as type promotions and taking the address of arrays and functions.

5.3. Machine-independent optimisation

The next pass performs optimisations and transformations of the tree. Typical of the transforms: `&*x` and `*&x` are converted into `x`. Constant expressions are converted to constants in this pass.

5.4. Arithmetic rewrites

This is another machine-independent optimisation. Subtrees of add, subtract, and multiply of integers are rewritten for easier compilation. The major transformation is factoring; `4+8*a+16*b+5` is transformed into `9+8*(a+2*b)`. Such expressions arise from address manipulation and array indexing.

5.5. Addressability

This is the first of the machine-dependent passes. The addressability of a computer is defined as the expression that is legal in the address field of a machine language instruction. The addressability of different computers varies widely. At one end of the spectrum are the 68020 and VAX, which allow a complex array of incrementing, decrementing, indexing and relative addressing. At the other end is the MIPS, which allows registers and constant offsets from the contents of a register. The addressability can be different for different instructions within the same computer.

It is important to the code generator to know when a subtree represents an address of a particular type. This

is done with a bottom-up walk of the tree. In this pass, the leaves are labelled with small integers. When an internal node is encountered, it is labelled by consulting a table indexed by the labels on the left and right subtrees. For example, on the 68020 computer, it is possible to address an offset from a named location. In C, this is represented by the expression `*(&name+constant)`. This is marked addressable by the following table. In the table, a node represented by the left column is marked with a small integer from the right column. Marks of the form `A1` are addressable while marks of the form `N1` are not addressable.

Node	**Marked**
`name`	`A1`
`const`	`A2`
`&A1`	`A3`
`A3+A1`	`N1` (note this is not addressable)
`*N1`	`A4`

Here there is a distinction between a node marked `A1` and a node marked `A4` because the address operator of an `A4` node is not addressable. So to extend the table:

Node	**Marked**
`&A4`	`N2`
`N2+N1`	`N1`

The full addressability of the 68020 is expressed in 18 rules like this. When one ports the compiler, this table is usually initialised so that leaves are labelled as addressable and nothing else. The code produced is poor, but porting is easy. The table can be extended later.

In the same bottom-up pass of the tree, the nodes are labelled with a Sethi-Ullman complexity. [Set70] This number is roughly the number of registers required to compile the tree on an ideal machine. An addressable node is marked 0. A function call is marked infinite. A unary operator is marked as the maximum of 1 and the mark of its subtree. A binary operator with equal marks on its subtrees is marked with a subtree mark plus 1. A binary operator with unequal marks on its subtrees is marked with the maximum mark of its subtrees. The actual values of the marks are not too important, but the relative values are. The goal is to compile the harder (larger mark) subtree first.

5.6. Code generation

Code is generated by simple recursive descent. The Sethi-Ullman complexity completely guides the order. The addressability defines the leaves. The only difficult part is compiling a tree that has two infinite (function call) subtrees. In this case, one subtree is compiled into the return register (usually the most convenient place for a function call) and then stored on the stack. The other subtree is compiled into the return register and then the operation is compiled with operands from the stack and the return register.

There is a separate boolean code generator that compiles conditional jumps. This is fundamentally different than compiling an expression. The result of the boolean code generator is the position of the program counter and not an expression. The boolean code generator is an expanded version of that described in chapter 8 of Aho, Sethi, and Ullman. [Aho87]

There is a considerable amount of talk in the literature about automating this part of a compiler with a machine description. Since this code generator is so small (less than 500 lines of C) and easy, it hardly seems worth the effort.

5.7. Registerisation

Up to now, the compiler has operated on syntax trees that are roughly equivalent to the original source language. The previous pass has produced machine language in an internal format. The next two passes operate on the internal machine language structures. The purpose of the next pass is to reintroduce registers for heavily used variables.

All of the variables that can be potentially registerised within a routine are placed in a table. (Suitable variables are all automatic or external scalars that do not have their addresses extracted. Some constants that are hard to reference are also considered for registerisation.) Four separate data flow equations are evaluated over the routine on all of these variables. Two of the equations are the normal set-behind and used-

ahead bits that define the life of a variable. The two new bits tell if a variable life crosses a function call ahead or behind. By examining a variable over its lifetime, it is possible to get a cost for registerising. Loops are detected and the costs are multiplied by three for every level of loop nesting. Costs are sorted and the variables are replaced by available registers on a greedy basis.

The 68020 has two different types of registers. For the 68020, two different costs are calculated for each variable life and the register type that affords the better cost is used. Ties are broken by counting the number of available registers of each type.

Note that externals are registerised together with automatics. This is done by evaluating the semantics of a ‘‘call’’ instruction differently for externals and automatics. Since a call goes outside the local procedure, it is assumed that a call references all externals. Similarly, externals are assumed to be set before an ‘‘entry’’ instruction and assumed to be referenced after a ‘‘return’’ instruction. This makes sure that externals are in memory across calls.

The overall results are very satisfying. It would be nice to be able to do this processing in a machine-independent way, but it is impossible to get all of the costs and side effects of different choices by examining the parse tree.

Most of the code in the registerisation pass is machine-independent. The major machine-dependency is in examining a machine instruction to ask if it sets or references a variable.

5.8. Machine code optimisation

The next pass walks the machine code for opportunistic optimisations. For the most part, this is highly specific to a particular computer. One optimisation that is performed on all of the computers is the removal of unnecessary ‘‘move’’ instructions. Ironically, most of these instructions were inserted by the previous pass. There are two patterns that are repetitively matched and replaced until no more matches are found. The first tries to remove ‘‘move’’ instructions by relabelling variables.

When a ‘‘move’’ instruction is encountered, if the destination variable is set before the source variable is referenced, then all of the references to the destination variable can be renamed to the source and the ‘‘move’’ can be deleted. This transformation uses the reverse data flow set up in the previous pass.

An example if this pattern is depicted in the following table. The pattern is in the left column and the replacement action is in the right column.

`MOVE a,b`	(remove)
(no use of `a`)	
`USE b`	`USE a`
(no use of `a`)	
`SET b`	`SET b`

Experiments have shown that it is marginally worth while to rename uses of the destination variable with uses of the source variable up to the first use of the source variable.

The second transform will do relabelling without deleting instructions. When a ‘‘move’’ instruction is encountered, if the source variable has been set prior to the use of the destination variable then all of the references to the source variable are replaced by the destination and the ‘‘move’’ is inverted. Typically, this transformation will alter two ‘‘move’’ instructions and allow the first transformation another chance to remove code. This transformation uses the forward data flow set up in the previous pass.

Again, the following is a depiction of the transformation where the pattern is in the left column and the rewrite is in the right column.

`SET a`	`SET b`
(no use of `b`)	
`USE a`	`USE b`
(no use of `b`)	
`MOVE a,b`	`MOVE b,a`

Iterating these transformations will usually get rid of all redundant ‘‘move’’ instructions.

A problem with this organisation is that the costs of registerisation calculated in the previous pass must depend on how well this pass can detect and remove redundant instructions. Often, a fine candidate for registerisation is rejected because of the cost of instructions that are later removed. Perhaps the registerisation pass should discount a large percentage of a ‘‘move’’ instruction anticipating the effectiveness of this pass.

5.9. Writing the object file

The last pass walks the internal assembly language and writes the object file. The object file is reduces in size by about a factor of three with simple compression techniques. The most important aspect of the object file format is that it is machine-independent. All integer and floating numbers in the object code are converted to known formats and byte orders. This is important for Plan 9 because the compiler might be run on different computers.

6. The loader

The loader is a multiple pass program that reads object files and libraries and produces an executable binary. The loader also does some minimal optimisations and code rewriting. Many of the operations performed by the loader are machine-dependent.

The first pass of the loader reads the object modules into an internal data structure that looks like binary assembly language. As the instructions are read, unconditional branch instructions are removed. Conditional branch instructions are inverted to prevent the insertion of unconditional branches. The loader will also make a copy of a few instructions to remove an unconditional branch. An example of this appears in a later section.

The next pass allocates addresses for all external data. Typical of computers is the 68020 which can reference ±32K from an address register. The loader allocates the address register `A6` as the static pointer. The value placed in `A6` is the base of the data segment plus 32K. It is then cheap to reference all data in the first 64K of the data segment. External variables are allocated to the data segment with the smallest variables allocated first. If all of the data cannot fit into the first 64K of the data segment, then usually only a few large arrays need more expensive addressing modes.

For the MIPS computer, the loader makes a pass over the internal structures exchanging instructions to try to fill ‘‘delay slots’’ with useful work. (A delay slot on the MIPS is a euphemism for a timing bug that must be avoided by the compiler.) If a useful instruction cannot be found to fill a delay slot, the loader will insert ‘‘noop’’ instructions. This pass is very expensive and does not do a good job. About 20% of all instructions are in delay slots. About 50% of these are useful instructions and 50% are ‘‘noops.’’ The vendor supplied assembler does this job much more effectively filling about 80% of the delay slots with useful instructions.

On the 68020 computer, branch instructions come in a variety of sizes depending on the relative distance of the branch. Thus the size of branch instructions can be mutually dependent on each other. The loader uses a multiple pass algorithm to resolve the branch lengths. [Szy78] Initially, all branches are assumed minimal length. On each subsequent pass, the branches are reassessed and expanded if necessary. When no more expansions occur, the locations of the instructions in the text segment are known.

On the MIPS computer, all instructions are one size. A single pass over the instructions will determine the locations of all addresses in the text segment.

The last pass of the loader produces the executable binary. A symbol table and other tables are produced to help the debugger to interpret the binary symbolically.

The loader has source line numbers at its disposal, but the interpretation of these numbers relative to `#include` files is not done. The loader is also in a good position to perform some global optimisations, but this has not been exploited.

7. Performance

The following is a table of the source size of the various components of the compilers.

lines	module
409	machine-independent compiler headers
975	machine-independent compiler Yacc
5161	machine-independent compiler C
819	68020 compiler headers
6574	68020 compiler C
223	68020 loader headers
4350	68020 loader C
461	MIPS compiler headers
4820	MIPS compiler C
263	MIPS loader headers
4035	MIPS loader C
3236	Crisp compiler headers
2526	Crisp compiler C
132	Crisp loader headers
2256	Crisp loader C

The following table is timing of a test program that does Quine-McClusky boolean function minimisation. The test program is a single file of 907 lines of C that is dominated by bit-picking and sorting. The execution time does not significantly depend on library implementation. Since no other compiler runs on Plan 9, these tests were run on a single-processor MIPS 3000 computer with vendor supplied software. The optimiser in the vendor supplied compiler is reputed to be extremely good. Another compiler, *lcc*, is compared in this list. *Lcc* is another new and highly portable compiler jointly written at Bell Labs and Princeton. None of the compilers were tuned on this test.

1.0s	new cc compile time
0.5s	new cc load time
90.4s	new cc run time
1.6s	vendor cc compile time
0.1s	vendor cc load time
138.8s	vendor cc run time
4.0s	vendor cc −O compile time
0.1s	vendor cc −O load time
84.7s	vendor cc −O run time
1.6s	vendor lcc compile time
0.1s	vendor lcc load time
96.3s	vendor lcc run time

Although it was not possible to directly compare *gcc* to the new compiler, *lcc* typically compiles in 50% of the time of *gcc* and the object runs in 75% of the time of *gcc*. The original *pcc* compiler is also not directly compared. It is too slow in both compilation and runtime to compete with the above compilers. Since *pcc* has not been updated to accept ANSI function prototypes, it is also hard to find test programs to form a comparison.

8. Example

Here is a small example of a fragment of C code to be compiled on the 68020 compiler.

```
int   a[10];
void
f(void)
{
      int i;

      for(i=0; i<10; i++)
            a[i] = i;
}
```

The following is the tree of the assignment statement after all machine-independent passes. The numbers in angle brackets are addressabilities. Numbers 10 or larger are addressable. The addressability, 9, for the INDEX operation means addressable if its second operand is placed in an index register. The number in parentheses is the Sethi-Ullman complexity. The typing information is at the end of each line.

```
ASSIGN (1) long
      INDEX <9> long
            ADDR <12> *long
                  NAME "a" 0 <10> long
            NAME "i" -4 <11> *long
      NAME "i" -4 <11> long
```

The following is the 68020 machine language generated before the registerisation pass. Note that there is no assembly language in this compiler; this is a print of the internal form in the same sense as the previous tree is a print of that internal form.

Here is some explanation of notation: (SP) denotes an automatic variable; (SB) denotes an external variable; A7 is the stack pointer, $4 is a constant.

```
f:    TEXT
      SUBL  $4,A7
      CLRL  i(SP)
loop: MOVL  $10,R0
      CMPL  R0,i(SP)
      BLE   ret
      MOVL  i(SP),R0
      MOVL  i(SP),a(SB)(R0.L*4)
      ADDL  $1,i(SP)
      JMP   loop
ret:  ADDL  $4,A7
      RTS
```

The following is the code after all compiling passes, but before loading:

```
f:    TEXT
      SUBL  $4,A7
      CLRL  R1
loop: MOVL  $10,R0
      CMPL  R0,R1
      BLE   ret
      MOVL  R1,a(SB)(R1.L*4)
      ADDL  $1,R1
      JMP   loop
ret:  ADDL  $4,A7
      RTS
```

The following is the code produced by the loader. The only real difference is the expansion and inversion

of the loop condition to prevent an unconditional branch.

```
f:    TEXT
      CLRL  R1
loop: MOVL  $10,R0
      CMPL  R0,R1
      BLE   ret
l1:   MOVL  R1,a(SB)(R1.L*4)
      ADDL  $1,R1
      MOVL  $10,R0
      CMPL  R0,R1
      BGT   l1
ret:  RTS
```

The compare sequence

```
      MOVL  $10,R0
      CMPL  R0,R1
```

was expanded from the single instruction

```
      CMPL  $10,R1
```

because the former is both shorter and faster. The relatively dumb loader made a second copy of the sequence without realising that the

```
      MOVL  $10,R0
```

is redundant.

9. Conclusions

The new compilers compile quickly, load slowly, and produce medium quality object code. The compilers are relatively portable, requiring but a couple weeks work to produce a compiler for a different computer. As a whole, the experiment is a success. For Plan 9, where we needed several compilers with specialised features and our own object formats, the project was indispensable.

Two problems have come up in retrospect. The first has to do with the division of labour between compiler and loader. Plan 9 runs on a multi-processor and as such compilations are often done in parallel. Unfortunately, all compilations must be complete before loading can begin. The load is then single-threaded. With this model, any shift of work from compile to load results in a significant increase in real time. The same is true of libraries that are compiled infrequently and loaded often. In the future, we will try to put some of the loader work back into the compiler.

The second problem comes from the various optimisations performed over several passes. Often optimisations in different passes depend on each other. Iterating the passes could compromise efficiency, or even loop. We see no real solution to this problem.

10. References

Aho87. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison Wesley, Reading, MA (1987).

Joh79a. S. C. Johnson, ‘‘A Tour Through the Portable C Compiler,’’ in *UNIX Programmer’s Manual, Seventh Ed., Vol. 2A*, AT&T Bell Laboratories, Murray Hill, NJ (1979).

Joh79b. S. C. Johnson, ‘‘YACC – Yet Another Compiler Compiler,’’ in *UNIX Programmer’s Manual, Seventh Ed., Vol. 2A*, AT&T Bell Laboratories, Murray Hill, NJ (1979).

Kan88. Gerry Kane, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ (1988).

Ker88. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ (1988).

Mot85. Motorola, *MC68020 32-Bit Microprocessor User's Manual, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ (1985).

Pik90. Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ''Plan 9 from Bell Labs,'' *Proc. UKUUG Conf.*, London, UK (July 1990).

Set70. R. Sethi and J. D. Ullman, ''The Generation of Optimal Code for Arithmetic Expressions,'' *J. ACM* **17**(4), pp. 715-728 (1970).

Szy78. T. G. Szymanski, ''Assembling Code for Machines with Span-dependent Instructions,'' *Comm. ACM* **21**(4), pp. 300-308 (1978).

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

[1] UNIX is a trademark of AT&T Bell Laboratories.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
Here are some simple transliterations to allow
    a non-C programmer to read this code.
=       assignment
==      equal to .EQ.
!=      not equal to .NE.
++      increment
'x'     single character constant
"xxx"   multiple character string
%d      format to convert to decimal
%s      format to convert to string
\t      tab character
\n      newline character
```

FIGURE 1.

STAGE II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this case, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

"Hello world\n"

represents a string with the character "\n," representing the new line character.

Figure 2.1 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge.

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2.1 is obvious and is presented in Figure 2.2. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 2.3. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 2.2.

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

STAGE III

Again, in the C compiler, Figure 3.1 represents the high level control of the C compiler where the routine "com-

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2.2.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

pile" is called to compile the next line of source. Figure 3.2 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

The final step is represented in Figure 3.3. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

MORAL

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity, (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heros of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punishable by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

Acknowledgment. I first read of the possibility of such a Trojan horse in an Air Force critique [4] of the security of an early implementation of Multics. I cannot find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

Plan 9 from Bell Labs

Rob Pike
Dave Presotto
Sean Dorward
Bob Flandrena
Ken Thompson
Howard Trickey
Phil Winterbottom

Bell Laboratories
Murray Hill, New Jersey 07974
USA

Motivation

By the mid 1980's, the trend in computing was away from large centralized time-shared computers towards networks of smaller, personal machines, typically UNIX 'workstations'. People had grown weary of overloaded, bureaucratic timesharing machines and were eager to move to small, self-maintained systems, even if that meant a net loss in computing power. As microcomputers became faster, even that loss was recovered, and this style of computing remains popular today.

In the rush to personal workstations, though, some of their weaknesses were overlooked. First, the operating system they run, UNIX, is itself an old timesharing system and has had trouble adapting to ideas born after it. Graphics and networking were added to UNIX well into its lifetime and remain poorly integrated and difficult to administer. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems. The choice of an old timesharing operating system to run those personal machines made it difficult to bind things together smoothly.

Plan 9 began in the late 1980's as an attempt to have it both ways: to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a time-sharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates. The philosophy is much like that of the Cambridge Distributed System [NeHe82]. The early catch phrase was to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes.

The problems with UNIX were too deep to fix, but some of its ideas could be brought along. The best was its use of the file system to coordinate naming of and

Appeared in a slightly different form in *Computing Systems,* Vol 8 #3, Summer 1995, pp. 221-254.

access to resources, even those, such as devices, not traditionally treated as files. For Plan 9, we adopted this idea by designing a network-level protocol, called 9P, to enable machines to access files on remote systems. Above this, we built a naming system that lets people and their computing agents build customized views of the resources in the network. This is where Plan 9 first began to look different: a Plan 9 user builds a private computing environment and recreates it wherever desired, rather than doing all computing on a private machine. It soon became clear that this model was richer than we had foreseen, and the ideas of per-process name spaces and file-system-like resources were extended throughout the system—to processes, graphics, even the network itself.

By 1989 the system had become solid enough that some of us began using it as our exclusive computing environment. This meant bringing along many of the services and applications we had used on UNIX. We used this opportunity to revisit many issues, not just kernel-resident ones, that we felt UNIX addressed badly. Plan 9 has new compilers, languages, libraries, window systems, and many new applications. Many of the old tools were dropped, while those brought along have been polished or rewritten.

Why be so all-encompassing? The distinction between operating system, library, and application is important to the operating system researcher but uninteresting to the user. What matters is clean functionality. By building a complete new system, we were able to solve problems where we thought they should be solved. For example, there is no real 'tty driver' in the kernel; that is the job of the window system. In the modern world, multi-vendor and multi-architecture computing are essential, yet the usual compilers and tools assume the program is being built to run locally; we needed to rethink these issues. Most important, though, the test of a system is the computing environment it provides. Producing a more efficient way to run the old UNIX warhorses is empty engineering; we were more interested in whether the new ideas suggested by the architecture of the underlying system encourage a more effective way of working. Thus, although Plan 9 provides an emulation environment for running POSIX commands, it is a backwater of the system. The vast majority of system software is developed in the 'native' Plan 9 environment.

There are benefits to having an all-new system. First, our laboratory has a history of building experimental peripheral boards. To make it easy to write device drivers, we want a system that is available in source form (no longer guaranteed with UNIX, even in the laboratory in which it was born). Also, we want to redistribute our work, which means the software must be locally produced. For example, we could have used some vendors' C compilers for our system, but even had we overcome the problems with cross-compilation, we would have difficulty redistributing the result.

This paper serves as an overview of the system. It discusses the architecture from the lowest building blocks to the computing environment seen by users. It also serves as an introduction to the rest of the Plan 9 Programmer's Manual, which it accompanies. More detail about topics in this paper can be found elsewhere in the manual.

Design

The view of the system is built upon three principles. First, resources are named and accessed like files in a hierarchical file system. Second, there is a standard protocol, called 9P, for accessing these resources. Third, the disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space. The unusual properties of Plan 9 stem from the consistent, aggressive application of these principles.

A large Plan 9 installation has a number of computers networked together, each providing a particular class of service. Shared multiprocessor servers provide computing cycles; other large machines offer file storage. These machines are located in an air-conditioned machine room and are connected by high-performance networks. Lower bandwidth networks such as Ethernet or ISDN connect these servers to office-

and home-resident workstations or PCs, called terminals in Plan 9 terminology. Figure 1 shows the arrangement.

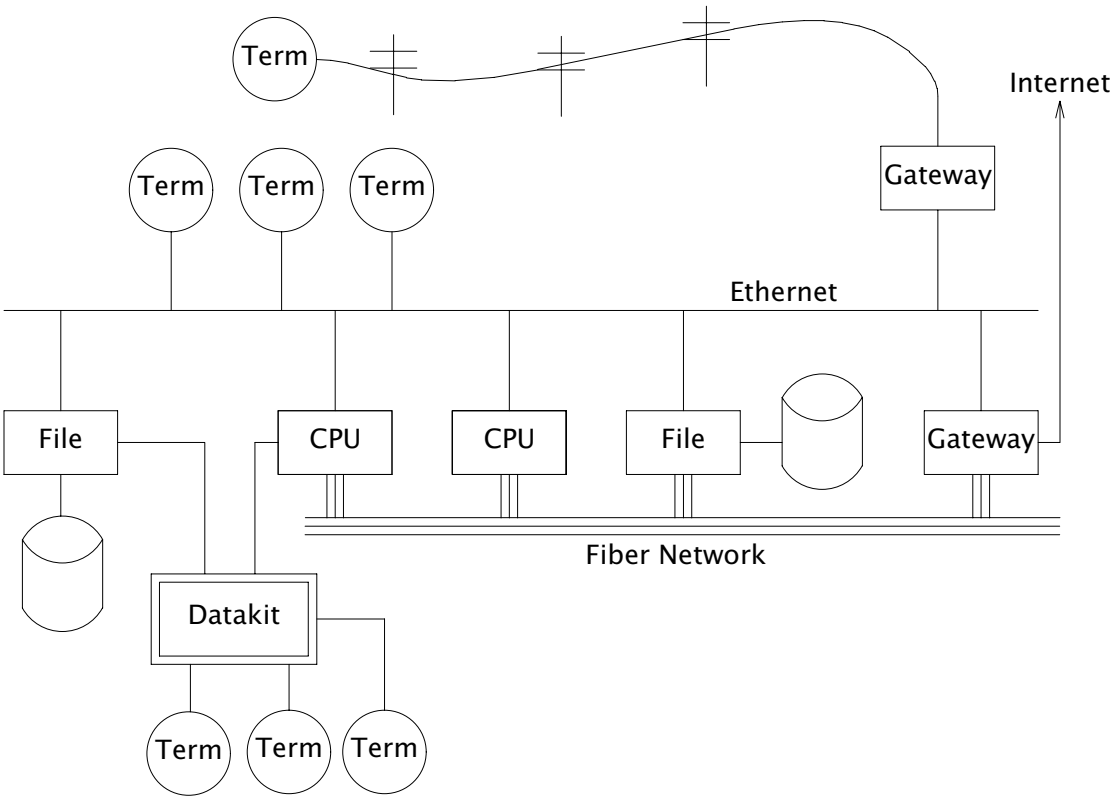


Figure 1. Structure of a large Plan 9 installation. CPU servers and file servers share fast local-area networks, while terminals use slower wider-area networks such as Ethernet, Datakit, or telephone lines to connect to them. Gateway machines, which are just CPU servers connected to multiple networks, allow machines on one network to see another.

The modern style of computing offers each user a dedicated workstation or PC. Plan 9's approach is different. The various machines with screens, keyboards, and mice all provide access to the resources of the network, so they are functionally equivalent, in the manner of the terminals attached to old timesharing systems. When someone uses the system, though, the terminal is temporarily personalized by that user. Instead of customizing the hardware, Plan 9 offers the ability to customize one's view of the system provided by the software. That customization is accomplished by giving local, personal names for the publicly visible resources in the network. Plan 9 provides the mechanism to assemble a personal view of the public space with local names for globally accessible resources. Since the most important resources of the network are files, the model of that view is file-oriented.

The client's local name space provides a way to customize the user's view of the network. The services available in the network all export file hierarchies. Those important to the user are gathered together into a custom name space; those of no immediate interest are ignored. This is a different style of use from the idea of a 'uniform global name space'. In Plan 9, there are known names for services and uniform names for files exported by those services, but the view is entirely local. As an analogy, consider the difference between the phrase 'my house' and the precise address of the speaker's home. The latter may be used by anyone but the former is easier to say and makes sense when spoken. It also changes meaning depending on who says it, yet that does

not cause confusion. Similarly, in Plan 9 the name /dev/cons always refers to the user's terminal and /bin/date the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing date. Plan 9, then, has local name spaces that obey globally understood conventions; it is the conventions that guarantee sane behavior in the presence of local names.

The 9P protocol is structured as a set of transactions that send a request from a client to a (local or remote) server and return the result. 9P controls file systems, not just files: it includes procedures to resolve file names and traverse the name hierarchy of the file system provided by the server. On the other hand, the client's name space is held by the client system alone, not on or with the server, a distinction from systems such as Sprite [OCDNW88]. Also, file access is at the level of bytes, not blocks, which distinguishes 9P from protocols like NFS and RFS. A paper by Welch compares Sprite, NFS, and Plan 9's network file system structures [Welc94].

This approach was designed with traditional files in mind, but can be extended to many other resources. Plan 9 services that export file hierarchies include I/O devices, backup services, the window system, network interfaces, and many others. One example is the process file system, /proc, which provides a clean way to examine and control running processes. Precursor systems had a similar idea [Kill84], but Plan 9 pushes the file metaphor much further [PPTTW93]. The file system model is well-understood, both by system builders and general users, so services that present file-like interfaces are easy to build, easy to understand, and easy to use. Files come with agreed-upon rules for protection, naming, and access both local and remote, so services built this way are ready-made for a distributed system. (This is a distinction from 'object-oriented' models, where these issues must be faced anew for every class of object.) Examples in the sections that follow illustrate these ideas in action.

The Command-level View

Plan 9 is meant to be used from a machine with a screen running the window system. It has no notion of 'teletype' in the UNIX sense. The keyboard handling of the bare system is rudimentary, but once the window system, 8½ [Pike91], is running, text can be edited with 'cut and paste' operations from a pop-up menu, copied between windows, and so on. 8½ permits editing text from the past, not just on the current input line. The text-editing capabilities of 8½ are strong enough to displace special features such as history in the shell, paging and scrolling, and mail editors. 8½ windows do not support cursor addressing and, except for one terminal emulator to simplify connecting to traditional systems, there is no cursor-addressing software in Plan 9.

Each window is created in a separate name space. Adjustments made to the name space in a window do not affect other windows or programs, making it safe to experiment with local modifications to the name space, for example to substitute files from the dump file system when debugging. Once the debugging is done, the window can be deleted and all trace of the experimental apparatus is gone. Similar arguments apply to the private space each window has for environment variables, notes (analogous to UNIX signals), etc.

Each window is created running an application, such as the shell, with standard input and output connected to the editable text of the window. Each window also has a private bitmap and multiplexed access to the keyboard, mouse, and other graphical resources through files like /dev/mouse, /dev/bitblt, and /dev/cons (analogous to UNIX's /dev/tty). These files are provided by 8½, which is implemented as a file server. Unlike X windows, where a new application typically creates a new window to run in, an 8½ graphics application usually runs in the window where it starts. It is possible and efficient for an application to create a new window, but that is not the style of the system. Again contrasting to X, in which a remote application makes a network call

to the X server to start running, a remote 8½ application sees the `mouse`, `bitblt`, and `cons` files for the window as usual in `/dev`; it does not know whether the files are local. It just reads and writes them to control the window; the network connection is already there and multiplexed.

The intended style of use is to run interactive applications such as the window system and text editor on the terminal and to run computation- or file-intensive applications on remote servers. Different windows may be running programs on different machines over different networks, but by making the name space equivalent in all windows, this is transparent: the same commands and resources are available, with the same names, wherever the computation is performed.

The command set of Plan 9 is similar to that of UNIX. The commands fall into several broad classes. Some are new programs for old jobs: programs like `ls`, `cat`, and `who` have familiar names and functions but are new, simpler implementations. `Who`, for example, is a shell script, while `ps` is just 95 lines of C code. Some commands are essentially the same as their UNIX ancestors: `awk`, `troff`, and others have been converted to ANSI C and extended to handle Unicode, but are still the familiar tools. Some are entirely new programs for old niches: the shell `rc`, text editor `sam`, debugger `acid`, and others displace the better-known UNIX tools with similar jobs. Finally, about half the commands are new.

Compatibility was not a requirement for the system. Where the old commands or notation seemed good enough, we kept them. When they didn't, we replaced them.

The File Server

A central file server stores permanent files and presents them to the network as a file hierarchy exported using 9P. The server is a stand-alone system, accessible only over the network, designed to do its one job well. It runs no user processes, only a fixed set of routines compiled into the boot image. Rather than a set of disks or separate file systems, the main hierarchy exported by the server is a single tree, representing files on many disks. That hierarchy is shared by many users over a wide area on a variety of networks. Other file trees exported by the server include special-purpose systems such as temporary storage and, as explained below, a backup service.

The file server has three levels of storage. The central server in our installation has about 100 megabytes of memory buffers, 27 gigabytes of magnetic disks, and 350 gigabytes of bulk storage in a write-once-read-many (WORM) jukebox. The disk is a cache for the WORM and the memory is a cache for the disk; each is much faster, and sees about an order of magnitude more traffic, than the level it caches. The addressable data in the file system can be larger than the size of the magnetic disks, because they are only a cache; our main file server has about 40 gigabytes of active storage.

The most unusual feature of the file server comes from its use of a WORM device for stable storage. Every morning at 5 o'clock, a *dump* of the file system occurs automatically. The file system is frozen and all blocks modified since the last dump are queued to be written to the WORM. Once the blocks are queued, service is restored and the read-only root of the dumped file system appears in a hierarchy of all dumps ever taken, named by its date. For example, the directory `/n/dump/1995/0315` is the root directory of an image of the file system as it appeared in the early morning of March 15, 1995. It takes a few minutes to queue the blocks, but the process to copy blocks to the WORM, which runs in the background, may take hours.

There are two ways the dump file system is used. The first is by the users themselves, who can browse the dump file system directly or attach pieces of it to their name space. For example, to track down a bug, it is straightforward to try the compiler from three months ago or to link a program with yesterday's library. With daily snapshots of all files, it is easy to find when a particular change was made or what changes were

made on a particular date. People feel free to make large speculative changes to files in the knowledge that they can be backed out with a single copy command. There is no backup system as such; instead, because the dump is in the file name space, backup problems can be solved with standard tools such as `cp`, `ls`, `grep`, and `diff`.

The other (very rare) use is complete system backup. In the event of disaster, the active file system can be initialized from any dump by clearing the disk cache and setting the root of the active file system to be a copy of the dumped root. Although easy to do, this is not to be taken lightly: besides losing any change made after the date of the dump, this recovery method results in a very slow system. The cache must be reloaded from WORM, which is much slower than magnetic disks. The file system takes a few days to reload the working set and regain its full performance.

Access permissions of files in the dump are the same as they were when the dump was made. Normal utilities have normal permissions in the dump without any special arrangement. The dump file system is read-only, though, which means that files in the dump cannot be written regardless of their permission bits; in fact, since directories are part of the read-only structure, even the permissions cannot be changed.

Once a file is written to WORM, it cannot be removed, so our users never see ''please clean up your files'' messages and there is no `df` command. We regard the WORM jukebox as an unlimited resource. The only issue is how long it will take to fill. Our WORM has served a community of about 50 users for five years and has absorbed daily dumps, consuming a total of 65% of the storage in the jukebox. In that time, the manufacturer has improved the technology, doubling the capacity of the individual disks. If we were to upgrade to the new media, we would have more free space than in the original empty jukebox. Technology has created storage faster than we can use it.

Unusual file servers

Plan 9 is characterized by a variety of servers that offer a file-like interface to unusual services. Many of these are implemented by user-level processes, although the distinction is unimportant to their clients; whether a service is provided by the kernel, a user process, or a remote server is irrelevant to the way it is used. There are dozens of such servers; in this section we present three representative ones.

Perhaps the most remarkable file server in Plan 9 is 8½, the window system. It is discussed at length elsewhere [Pike91], but deserves a brief explanation here. 8½ provides two interfaces: to the user seated at the terminal, it offers a traditional style of interaction with multiple windows, each running an application, all controlled by a mouse and keyboard. To the client programs, the view is also fairly traditional: programs running in a window see a set of files in `/dev` with names like `mouse`, `screen`, and `cons`. Programs that want to print text to their window write to `/dev/cons`; to read the mouse, they read `/dev/mouse`. In the Plan 9 style, bitmap graphics is implemented by providing a file `/dev/bitblt` on which clients write encoded messages to execute graphical operations such as `bitblt` (RasterOp). What is unusual is how this is done: 8½ is a file server, serving the files in `/dev` to the clients running in each window. Although every window looks the same to its client, each window has a distinct set of files in `/dev`. 8½ multiplexes its clients' access to the resources of the terminal by serving multiple sets of files. Each client is given a private name space with a *different* set of files that behave the same as in all other windows. There are many advantages to this structure. One is that 8½ serves the same files it needs for its own implementation—it multiplexes its own interface—so it may be run, recursively, as a client of itself. Also, consider the implementation of `/dev/tty` in UNIX, which requires special code in the kernel to redirect `open` calls to the appropriate device. Instead, in 8½ the equivalent service falls out automatically: 8½ serves `/dev/cons` as its basic function; there is nothing extra to do. When a program wants to read from the keyboard, it opens `/dev/cons`, but it is a private file, not a shared one with special

properties. Again, local name spaces make this possible; conventions about the consistency of the files within them make it natural.

8½ has a unique feature made possible by its design. Because it is implemented as a file server, it has the power to postpone answering read requests for a particular window. This behavior is toggled by a reserved key on the keyboard. Toggling once suspends client reads from the window; toggling again resumes normal reads, which absorb whatever text has been prepared, one line at a time. This allows the user to edit multi-line input text on the screen before the application sees it, obviating the need to invoke a separate editor to prepare text such as mail messages. A related property is that reads are answered directly from the data structure defining the text on the display: text may be edited until its final newline makes the prepared line of text readable by the client. Even then, until the line is read, the text the client will read can be changed. For example, after typing

```
% make
rm *
```

to the shell, the user can backspace over the final newline at any time until `make` finishes, holding off execution of the `rm` command, or even point with the mouse before the `rm` and type another command to be executed first.

There is no `ftp` command in Plan 9. Instead, a user-level file server called `ftpfs` dials the FTP site, logs in on behalf of the user, and uses the FTP protocol to examine files in the remote directory. To the local user, it offers a file hierarchy, attached to `/n/ftp` in the local name space, mirroring the contents of the FTP site. In other words, it translates the FTP protocol into 9P to offer Plan 9 access to FTP sites. The implementation is tricky; `ftpfs` must do some sophisticated caching for efficiency and use heuristics to decode remote directory information. But the result is worthwhile: all the local file management tools such as `cp`, `grep`, `diff`, and of course `ls` are available to FTP-served files exactly as if they were local files. Other systems such as Jade and Prospero have exploited the same opportunity [Rao81, Neu92], but because of local name spaces and the simplicity of implementing 9P, this approach fits more naturally into Plan 9 than into other environments.

One server, `exportfs`, is a user process that takes a portion of its own name space and makes it available to other processes by translating 9P requests into system calls to the Plan 9 kernel. The file hierarchy it exports may contain files from multiple servers. `Exportfs` is usually run as a remote server started by a local program, either `import` or `cpu`. `Import` makes a network call to the remote machine, starts `exportfs` there, and attaches its 9P connection to the local name space. For example,

```
import helix /net
```

makes Helix's network interfaces visible in the local `/net` directory. Helix is a central server and has many network interfaces, so this permits a machine with one network to access to any of Helix's networks. After such an import, the local machine may make calls on any of the networks connected to Helix. Another example is

```
import helix /proc
```

which makes Helix's processes visible in the local `/proc`, permitting local debuggers to examine remote processes.

The `cpu` command connects the local terminal to a remote CPU server. It works in the opposite direction to `import`: after calling the server, it starts a *local* `exportfs` and mounts it in the name space of a process, typically a newly created shell, on the server. It then rearranges the name space to make local device files (such as those served by the terminal's window system) visible in the server's `/dev` directory. The effect of running a `cpu` command is therefore to start a shell on a fast machine, one more tightly coupled to the file server, with a name space analogous to the local one.

All local device files are visible remotely, so remote applications have full access to local services such as bitmap graphics, /dev/cons, and so on. This is not the same as rlogin, which does nothing to reproduce the local name space on the remote system, nor is it the same as file sharing with, say, NFS, which can achieve some name space equivalence but not the combination of access to local hardware devices, remote files, and remote CPU resources. The cpu command is a uniquely transparent mechanism. For example, it is reasonable to start a window system in a window running a cpu command; all windows created there automatically start processes on the CPU server.

Configurability and administration

The uniform interconnection of components in Plan 9 makes it possible to configure a Plan 9 installation many different ways. A single laptop PC can function as a stand-alone Plan 9 system; at the other extreme, our setup has central multiprocessor CPU servers and file servers and scores of terminals ranging from small PCs to high-end graphics workstations. It is such large installations that best represent how Plan 9 operates.

The system software is portable and the same operating system runs on all hardware. Except for performance, the appearance of the system on, say, an SGI workstation is the same as on a laptop. Since computing and file services are centralized, and terminals have no permanent file storage, all terminals are functionally identical. In this way, Plan 9 has one of the good properties of old timesharing systems, where a user could sit in front of any machine and see the same system. In the modern workstation community, machines tend to be owned by people who customize them by storing private information on local disk. We reject this style of use, although the system itself can be used this way. In our group, we have a laboratory with many public-access machines—a terminal room—and a user may sit down at any one of them and work.

Central file servers centralize not just the files, but also their administration and maintenance. In fact, one server is the main server, holding all system files; other servers provide extra storage or are available for debugging and other special uses, but the system software resides on one machine. This means that each program has a single copy of the binary for each architecture, so it is trivial to install updates and bug fixes. There is also a single user database; there is no need to synchronize distinct /etc/passwd files. On the other hand, depending on a single central server does limit the size of an installation.

Another example of the power of centralized file service is the way Plan 9 administers network information. On the central server there is a directory, /lib/ndb, that contains all the information necessary to administer the local Ethernet and other networks. All the machines use the same database to talk to the network; there is no need to manage a distributed naming system or keep parallel files up to date. To install a new machine on the local Ethernet, choose a name and IP address and add these to a single file in /lib/ndb; all the machines in the installation will be able to talk to it immediately. To start running, plug the machine into the network, turn it on, and use BOOTP and TFTP to load the kernel. All else is automatic.

Finally, the automated dump file system frees all users from the need to maintain their systems, while providing easy access to backup files without tapes, special commands, or the involvement of support staff. It is difficult to overstate the improvement in lifestyle afforded by this service.

Plan 9 runs on a variety of hardware without constraining how to configure an installation. In our laboratory, we chose to use central servers because they amortize costs and administration. A sign that this is a good decision is that our cheap terminals remain comfortable places to work for about five years, much longer than workstations that must provide the complete computing environment. We do, however, upgrade the central machines, so the computation available from even old Plan 9 terminals improves

with time. The money saved by avoiding regular upgrades of terminals is instead spent on the newest, fastest multiprocessor servers. We estimate this costs about half the money of networked workstations yet provides general access to more powerful machines.

C Programming

Plan 9 utilities are written in several languages. Some are scripts for the shell, `rc` [Duff90]; a handful are written in a new C-like concurrent language called Alef [Wint95], described below. The great majority, though, are written in a dialect of ANSI C [ANSIC]. Of these, most are entirely new programs, but some originate in pre-ANSI C code from our research UNIX system [UNIX85]. These have been updated to ANSI C and reworked for portability and cleanliness.

The Plan 9 C dialect has some minor extensions, described elsewhere [Pike95], and a few major restrictions. The most important restriction is that the compiler demands that all function definitions have ANSI prototypes and all function calls appear in the scope of a prototyped declaration of the function. As a stylistic rule, the prototyped declaration is placed in a header file included by all files that call the function. Each system library has an associated header file, declaring all functions in that library. For example, the standard Plan 9 library is called `libc`, so all C source files include `<libc.h>`. These rules guarantee that all functions are called with arguments having the expected types — something that was not true with pre-ANSI C programs.

Another restriction is that the C compilers accept only a subset of the preprocessor directives required by ANSI. The main omission is `#if`, since we believe it is never necessary and often abused. Also, its effect is better achieved by other means. For instance, an `#if` used to toggle a feature at compile time can be written as a regular `if` statement, relying on compile-time constant folding and dead code elimination to discard object code.

Conditional compilation, even with `#ifdef`, is used sparingly in Plan 9. The only architecture-dependent `#ifdefs` in the system are in low-level routines in the graphics library. Instead, we avoid such dependencies or, when necessary, isolate them in separate source files or libraries. Besides making code hard to read, `#ifdefs` make it impossible to know what source is compiled into the binary or whether source protected by them will compile or work properly. They make it harder to maintain software.

The standard Plan 9 library overlaps much of ANSI C and POSIX [POSIX], but diverges when appropriate to Plan 9's goals or implementation. When the semantics of a function change, we also change the name. For instance, instead of UNIX's `creat`, Plan 9 has a `create` function that takes three arguments, the original two plus a third that, like the second argument of `open`, defines whether the returned file descriptor is to be opened for reading, writing, or both. This design was forced by the way 9P implements creation, but it also simplifies the common use of `create` to initialize a temporary file.

Another departure from ANSI C is that Plan 9 uses a 16-bit character set called Unicode [ISO10646, Unicode]. Although we stopped short of full internationalization, Plan 9 treats the representation of all major languages uniformly throughout all its software. To simplify the exchange of text between programs, the characters are packed into a byte stream by an encoding we designed, called UTF-8, which is now becoming accepted as a standard [FSSUTF]. It has several attractive properties, including byte-order independence, backwards compatibility with ASCII, and ease of implementation.

There are many problems in adapting existing software to a large character set with an encoding that represents characters with a variable number of bytes. ANSI C addresses some of the issues but falls short of solving them all. It does not pick a character set encoding and does not define all the necessary I/O library routines.

Furthermore, the functions it *does* define have engineering problems. Since the standard left too many problems unsolved, we decided to build our own interface. A separate paper has the details [Pike93].

A small class of Plan 9 programs do not follow the conventions discussed in this section. These are programs imported from and maintained by the UNIX community; `tex` is a representative example. To avoid reconverting such programs every time a new version is released, we built a porting environment, called the ANSI C/POSIX Environment, or APE [Tric95]. APE comprises separate include files, libraries, and commands, conforming as much as possible to the strict ANSI C and base-level POSIX specifications. To port network-based software such as X Windows, it was necessary to add some extensions to those specifications, such as the BSD networking functions.

Portability and Compilation

Plan 9 is portable across a variety of processor architectures. Within a single computing session, it is common to use several architectures: perhaps the window system running on an Intel processor connected to a MIPS-based CPU server with files resident on a SPARC system. For this heterogeneity to be transparent, there must be conventions about data interchange between programs; for software maintenance to be straightforward, there must be conventions about cross-architecture compilation.

To avoid byte order problems, data is communicated between programs as text whenever practical. Sometimes, though, the amount of data is high enough that a binary format is necessary; such data is communicated as a byte stream with a predefined encoding for multi-byte values. In the rare cases where a format is complex enough to be defined by a data structure, the structure is never communicated as a unit; instead, it is decomposed into individual fields, encoded as an ordered byte stream, and then reassembled by the recipient. These conventions affect data ranging from kernel or application program state information to object file intermediates generated by the compiler.

Programs, including the kernel, often present their data through a file system interface, an access mechanism that is inherently portable. For example, the system clock is represented by a decimal number in the file `/dev/time`; the `time` library function (there is no `time` system call) reads the file and converts it to binary. Similarly, instead of encoding the state of an application process in a series of flags and bits in private memory, the kernel presents a text string in the file named `status` in the `/proc` file system associated with each process. The Plan 9 `ps` command is trivial: it prints the contents of the desired status files after some minor reformatting; moreover, after

```
import helix /proc
```

a local `ps` command reports on the status of Helix's processes.

Each supported architecture has its own compilers and loader. The C and Alef compilers produce intermediate files that are portably encoded; the contents are unique to the target architecture but the format of the file is independent of compiling processor type. When a compiler for a given architecture is compiled on another type of processor and then used to compile a program there, the intermediate produced on the new architecture is identical to the intermediate produced on the native processor. From the compiler's point of view, every compilation is a cross-compilation.

Although each architecture's loader accepts only intermediate files produced by compilers for that architecture, such files could have been generated by a compiler executing on any type of processor. For instance, it is possible to run the MIPS compiler on a 486, then use the MIPS loader on a SPARC to produce a MIPS executable.

Since Plan 9 runs on a variety of architectures, even in a single installation, distinguishing the compilers and intermediate names simplifies multi-architecture

development from a single source tree. The compilers and the loader for each architecture are uniquely named; there is no `cc` command. The names are derived by concatenating a code letter associated with the target architecture with the name of the compiler or loader. For example, the letter '8' is the code letter for Intel *x*86 processors; the C compiler is named `8c`, the Alef compiler `8al`, and the loader is called `8l`. Similarly, the compiler intermediate files are suffixed `.8`, not `.o`.

The Plan 9 build program `mk`, a relative of `make`, reads the names of the current and target architectures from environment variables called `$cputype` and `$objtype`. By default the current processor is the target, but setting `$objtype` to the name of another architecture before invoking `mk` results in a cross-build:

```
% objtype=sparc mk
```

builds a program for the SPARC architecture regardless of the executing machine. The value of `$objtype` selects a file of architecture-dependent variable definitions that configures the build to use the appropriate compilers and loader. Although simple-minded, this technique works well in practice: all applications in Plan 9 are built from a single source tree and it is possible to build the various architectures in parallel without conflict.

Parallel programming

Plan 9's support for parallel programming has two aspects. First, the kernel provides a simple process model and a few carefully designed system calls for synchronization and sharing. Second, a new parallel programming language called Alef supports concurrent programming. Although it is possible to write parallel programs in C, Alef is the parallel language of choice.

There is a trend in new operating systems to implement two classes of processes: normal UNIX-style processes and light-weight kernel threads. Instead, Plan 9 provides a single class of process but allows fine control of the sharing of a process's resources such as memory and file descriptors. A single class of process is a feasible approach in Plan 9 because the kernel has an efficient system call interface and cheap process creation and scheduling.

Parallel programs have three basic requirements: management of resources shared between processes, an interface to the scheduler, and fine-grain process synchronization using spin locks. On Plan 9, new processes are created using the `rfork` system call. `Rfork` takes a single argument, a bit vector that specifies which of the parent process's resources should be shared, copied, or created anew in the child. The resources controlled by `rfork` include the name space, the environment, the file descriptor table, memory segments, and notes (Plan 9's analog of UNIX signals). One of the bits controls whether the `rfork` call will create a new process; if the bit is off, the resulting modification to the resources occurs in the process making the call. For example, a process calls `rfork(RFNAMEG)` to disconnect its name space from its parent's. Alef uses a fine-grained fork in which all the resources, including memory, are shared between parent and child, analogous to creating a kernel thread in many systems.

An indication that `rfork` is the right model is the variety of ways it is used. Other than the canonical use in the library routine `fork`, it is hard to find two calls to `rfork` with the same bits set; programs use it to create many different forms of sharing and resource allocation. A system with just two types of processes—regular processes and threads—could not handle this variety.

There are two ways to share memory. First, a flag to `rfork` causes all the memory segments of the parent to be shared with the child (except the stack, which is forked copy-on-write regardless). Alternatively, a new segment of memory may be attached using the `segattach` system call; such a segment will always be shared between parent and child.

The rendezvous system call provides a way for processes to synchronize. Alef uses it to implement communication channels, queuing locks, multiple reader/writer locks, and the sleep and wakeup mechanism. Rendezvous takes two arguments, a tag and a value. When a process calls rendezvous with a tag it sleeps until another process presents a matching tag. When a pair of tags match, the values are exchanged between the two processes and both rendezvous calls return. This primitive is sufficient to implement the full set of synchronization routines.

Finally, spin locks are provided by an architecture-dependent library at user level. Most processors provide atomic test and set instructions that can be used to implement locks. A notable exception is the MIPS R3000, so the SGI Power series multiprocessors have special lock hardware on the bus. User processes gain access to the lock hardware by mapping pages of hardware locks into their address space using the segattach system call.

A Plan 9 process in a system call will block regardless of its 'weight'. This means that when a program wishes to read from a slow device without blocking the entire calculation, it must fork a process to do the read for it. The solution is to start a satellite process that does the I/O and delivers the answer to the main program through shared memory or perhaps a pipe. This sounds onerous but works easily and efficiently in practice; in fact, most interactive Plan 9 applications, even relatively ordinary ones written in C, such as the text editor Sam [Pike87], run as multiprocess programs.

The kernel support for parallel programming in Plan 9 is a few hundred lines of portable code; a handful of simple primitives enable the problems to be handled cleanly at user level. Although the primitives work fine from C, they are particularly expressive from within Alef. The creation and management of slave I/O processes can be written in a few lines of Alef, providing the foundation for a consistent means of multiplexing data flows between arbitrary processes. Moreover, implementing it in a language rather than in the kernel ensures consistent semantics between all devices and provides a more general multiplexing primitive. Compare this to the UNIX select system call: select applies only to a restricted set of devices, legislates a style of multiprogramming in the kernel, does not extend across networks, is difficult to implement, and is hard to use.

Another reason parallel programming is important in Plan 9 is that multi-threaded user-level file servers are the preferred way to implement services. Examples of such servers include the programming environment Acme [Pike94], the name space exporting tool exportfs [PPTTW93], the HTTP daemon, and the network name servers cs and dns [PrWi93]. Complex applications such as Acme prove that careful operating system support can reduce the difficulty of writing multi-threaded applications without moving threading and synchronization primitives into the kernel.

Implementation of Name Spaces

User processes construct name spaces using three system calls: mount, bind, and unmount. The mount system call attaches a tree served by a file server to the current name space. Before calling mount, the client must (by outside means) acquire a connection to the server in the form of a file descriptor that may be written and read to transmit 9P messages. That file descriptor represents a pipe or network connection.

The mount call attaches a new hierarchy to the existing name space. The bind system call, on the other hand, duplicates some piece of existing name space at another point in the name space. The unmount system call allows components to be removed.

Using either bind or mount, multiple directories may be stacked at a single point in the name space. In Plan 9 terminology, this is a *union* directory and behaves like the concatenation of the constituent directories. A flag argument to bind and mount specifies the position of a new directory in the union, permitting new elements to be added either at the front or rear of the union or to replace it entirely. When a file lookup

is performed in a union directory, each component of the union is searched in turn and the first match taken; likewise, when a union directory is read, the contents of each of the component directories is read in turn. Union directories are one of the most widely used organizational features of the Plan 9 name space. For instance, the directory `/bin` is built as a union of `/$cputype/bin` (program binaries), `/rc/bin` (shell scripts), and perhaps more directories provided by the user. This construction makes the shell `$PATH` variable unnecessary.

One question raised by union directories is which element of the union receives a newly created file. After several designs, we decided on the following. By default, directories in unions do not accept new files, although the `create` system call applied to an existing file succeeds normally. When a directory is added to the union, a flag to `bind` or `mount` enables create permission (a property of the name space) in that directory. When a file is being created with a new name in a union, it is created in the first directory of the union with create permission; if that creation fails, the entire `create` fails. This scheme enables the common use of placing a private directory anywhere in a union of public ones, while allowing creation only in the private directory.

By convention, kernel device file systems are bound into the `/dev` directory, but to bootstrap the name space building process it is necessary to have a notation that permits direct access to the devices without an existing name space. The root directory of the tree served by a device driver can be accessed using the syntax *#c*, where *c* is a unique character (typically a letter) identifying the *type* of the device. Simple device drivers serve a single level directory containing a few files. As an example, each serial port is represented by a data and a control file:

```
% bind -a '#t' /dev
% cd /dev
% ls -l eia*
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia1
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia1ctl
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia2
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia2ctl
```

The `bind` program is an encapsulation of the `bind` system call; its `-a` flag positions the new directory at the end of the union. The data files `eia1` and `eia2` may be read and written to communicate over the serial line. Instead of using special operations on these files to control the devices, commands written to the files `eia1ctl` and `eia2ctl` control the corresponding device; for example, writing the text string `b1200` to `/dev/eia1ctl` sets the speed of that line to 1200 baud. Compare this to the UNIX `ioctl` system call: in Plan 9, devices are controlled by textual messages, free of byte order problems, with clear semantics for reading and writing. It is common to configure or debug devices using shell scripts.

It is the universal use of the 9P protocol that connects Plan 9's components together to form a distributed system. Rather than inventing a unique protocol for each service such as `rlogin`, FTP, TFTP, and X windows, Plan 9 implements services in terms of operations on file objects, and then uses a single, well-documented protocol to exchange information between computers. Unlike NFS, 9P treats files as a sequence of bytes rather than blocks. Also unlike NFS, 9P is stateful: clients perform remote procedure calls to establish pointers to objects in the remote file server. These pointers are called file identifiers or *fids*. All operations on files supply a fid to identify an object in the remote file system.

The 9P protocol defines 17 messages, providing means to authenticate users, navigate fids around a file system hierarchy, copy fids, perform I/O, change file attributes, and create and delete files. Its complete specification is in Section 5 of the Programmer's Manual [9man]. Here is the procedure to gain access to the name hierarchy supplied by a server. A file server connection is established via a pipe or network

connection. An initial `session` message performs a bilateral authentication between client and server. An `attach` message then connects a fid suggested by the client to the root of the server file tree. The `attach` message includes the identity of the user performing the attach; henceforth all fids derived from the root fid will have permissions associated with that user. Multiple users may share the connection, but each must perform an attach to establish his or her identity.

The `walk` message moves a fid through a single level of the file system hierarchy. The `clone` message takes an established fid and produces a copy that points to the same file as the original. Its purpose is to enable walking to a file in a directory without losing the fid on the directory. The `open` message locks a fid to a specific file in the hierarchy, checks access permissions, and prepares the fid for I/O. The `read` and `write` messages allow I/O at arbitrary offsets in the file; the maximum size transferred is defined by the protocol. The `clunk` message indicates the client has no further use for a fid. The `remove` message behaves like `clunk` but causes the file associated with the fid to be removed and any associated resources on the server to be deallocated.

9P has two forms: RPC messages sent on a pipe or network connection and a procedural interface within the kernel. Since kernel device drivers are directly addressable, there is no need to pass messages to communicate with them; instead each 9P transaction is implemented by a direct procedure call. For each fid, the kernel maintains a local representation in a data structure called a *channel*, so all operations on files performed by the kernel involve a channel connected to that fid. The simplest example is a user process's file descriptors, which are indexes into an array of channels. A table in the kernel provides a list of entry points corresponding one to one with the 9P messages for each device. A system call such as `read` from the user translates into one or more procedure calls through that table, indexed by the type character stored in the channel: `procread`, `eiaread`, etc. Each call takes at least one channel as an argument. A special kernel driver, called the *mount* driver, translates procedure calls to messages, that is, it converts local procedure calls to remote ones. In effect, this special driver becomes a local proxy for the files served by a remote file server. The channel pointer in the local call is translated to the associated fid in the transmitted message.

The mount driver is the sole RPC mechanism employed by the system. The semantics of the supplied files, rather than the operations performed upon them, create a particular service such as the `cpu` command. The mount driver demultiplexes protocol messages between clients sharing a communication channel with a file server. For each outgoing RPC message, the mount driver allocates a buffer labeled by a small unique integer, called a *tag*. The reply to the RPC is labeled with the same tag, which is used by the mount driver to match the reply with the request.

The kernel representation of the name space is called the *mount table*, which stores a list of bindings between channels. Each entry in the mount table contains a pair of channels: a *from* channel and a *to* channel. Every time a walk succeeds in moving a channel to a new location in the name space, the mount table is consulted to see if a 'from' channel matches the new name; if so the 'to' channel is cloned and substituted for the original. Union directories are implemented by converting the 'to' channel into a list of channels: a successful walk to a union directory returns a 'to' channel that forms the head of a list of channels, each representing a component directory of the union. If a walk fails to find a file in the first directory of the union, the list is followed, the next component cloned, and walk tried on that directory.

Each file in Plan 9 is uniquely identified by a set of integers: the type of the channel (used as the index of the function call table), the server or device number distinguishing the server from others of the same type (decided locally by the driver), and a *qid* formed from two 32-bit numbers called *path* and *version*. The path is a unique file number assigned by a device driver or file server when a file is created. The version number is updated whenever the file is modified; as described in the next section, it can be used

to maintain cache coherency between clients and servers.

The type and device number are analogous to UNIX major and minor device numbers; the qid is analogous to the i-number. The device and type connect the channel to a device driver and the qid identifies the file within that device. If the file recovered from a walk has the same type, device, and qid path as an entry in the mount table, they are the same file and the corresponding substitution from the mount table is made. This is how the name space is implemented.

File Caching

The 9P protocol has no explicit support for caching files on a client. The large memory of the central file server acts as a shared cache for all its clients, which reduces the total amount of memory needed across all machines in the network. Nonetheless, there are sound reasons to cache files on the client, such as a slow connection to the file server.

The version field of the qid is changed whenever the file is modified, which makes it possible to do some weakly coherent forms of caching. The most important is client caching of text and data segments of executable files. When a process `execs` a program, the file is re-opened and the qid's version is compared with that in the cache; if they match, the local copy is used. The same method can be used to build a local caching file server. This user-level server interposes on the 9P connection to the remote server and monitors the traffic, copying data to a local disk. When it sees a read of known data, it answers directly, while writes are passed on immediately—the cache is write-through—to keep the central copy up to date. This is transparent to processes on the terminal and requires no change to 9P; it works well on home machines connected over serial lines. A similar method can be applied to build a general client cache in unused local memory, but this has not been done in Plan 9.

Networks and Communication Devices

Network interfaces are kernel-resident file systems, analogous to the EIA device described earlier. Call setup and shutdown are achieved by writing text strings to the control file associated with the device; information is sent and received by reading and writing the data file. The structure and semantics of the devices is common to all networks so, other than a file name substitution, the same procedure makes a call using TCP over Ethernet as URP over Datakit [Fra80].

This example illustrates the structure of the TCP device:

```
% ls -lp /net/tcp
d-r-xr-xr-x I 0 bootes bootes 0 Feb 23 20:20 0
d-r-xr-xr-x I 0 bootes bootes 0 Feb 23 20:20 1
--rw-rw-rw- I 0 bootes bootes 0 Feb 23 20:20 clone
% ls -lp /net/tcp/0
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 ctl
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 data
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 listen
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 local
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 remote
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 status
%
```

The top directory, `/net/tcp`, contains a `clone` file and a directory for each connection, numbered 0 to *n*. Each connection directory corresponds to an TCP/IP connection. Opening `clone` reserves an unused connection and returns its control file. Reading the control file returns the textual connection number, so the user process can construct the full name of the newly allocated connection directory. The `local`, `remote`, and `status` files are diagnostic; for example, `remote` contains the address (for TCP, the

IP address and port number) of the remote side.

A call is initiated by writing a connect message with a network-specific address as its argument; for example, to open a Telnet session (port 23) to a remote machine with IP address 135.104.9.52, the string is:

```
connect 135.104.9.52!23
```

The write to the control file blocks until the connection is established; if the destination is unreachable, the write returns an error. Once the connection is established, the `telnet` application reads and writes the `data` file to talk to the remote Telnet daemon. On the other end, the Telnet daemon would start by writing

```
announce 23
```

to its control file to indicate its willingness to receive calls to this port. Such a daemon is called a *listener* in Plan 9.

A uniform structure for network devices cannot hide all the details of addressing and communication for dissimilar networks. For example, Datakit uses textual, hierarchical addresses unlike IP's 32-bit addresses, so an application given a control file must still know what network it represents. Rather than make every application know the addressing of every network, Plan 9 hides these details in a *connection server*, called `cs`. `Cs` is a file system mounted in a known place. It supplies a single control file that an application uses to discover how to connect to a host. The application writes the symbolic address and service name for the connection it wishes to make, and reads back the name of the `clone` file to open and the address to present to it. If there are multiple networks between the machines, `cs` presents a list of possible networks and addresses to be tried in sequence; it uses heuristics to decide the order. For instance, it presents the highest-bandwidth choice first.

A single library function called `dial` talks to `cs` to establish the connection. An application that uses `dial` needs no changes, not even recompilation, to adapt to new networks; the interface to `cs` hides the details.

The uniform structure for networks in Plan 9 makes the `import` command all that is needed to construct gateways.

Kernel structure for networks

The kernel plumbing used to build Plan 9 communications channels is called *streams* [Rit84][Presotto]. A stream is a bidirectional channel connecting a physical or pseudo-device to a user process. The user process inserts and removes data at one end of the stream; a kernel process acting on behalf of a device operates at the other end. A stream comprises a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Calling the put routine of the module on either end of the stream inserts data into the stream. Each module calls the succeeding one to send data up or down the stream. Like UNIX streams [Rit84], Plan 9 streams can be dynamically configured.

The IL Protocol

The 9P protocol must run above a reliable transport protocol with delimited messages. 9P has no mechanism to recover from transmission errors and the system assumes that each read from a communication channel will return a single 9P message; it does not parse the data stream to discover message boundaries. Pipes and some network protocols already have these properties but the standard IP protocols do not. TCP does not delimit messages, while UDP [RFC768] does not provide reliable in-order delivery.

We designed a new protocol, called IL (Internet Link), to transmit 9P messages over IP. It is a connection-based protocol that provides reliable transmission of sequenced

messages between machines. Since a process can have only a single outstanding 9P request, there is no need for flow control in IL. Like TCP, IL has adaptive timeouts: it scales acknowledge and retransmission times to match the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets. Also, IL does no blind retransmission, to avoid adding to the congestion of busy networks. Full details are in another paper [PrWi95].

In Plan 9, the implementation of IL is smaller and faster than TCP. IL is our main Internet transport protocol.

Overview of authentication

Authentication establishes the identity of a user accessing a resource. The user requesting the resource is called the *client* and the user granting access to the resource is called the *server*. This is usually done under the auspices of a 9P attach message. A user may be a client in one authentication exchange and a server in another. Servers always act on behalf of some user, either a normal client or some administrative entity, so authentication is defined to be between users, not machines.

Each Plan 9 user has an associated DES [NBS77] authentication key; the user's identity is verified by the ability to encrypt and decrypt special messages called challenges. Since knowledge of a user's key gives access to that user's resources, the Plan 9 authentication protocols never transmit a message containing a cleartext key.

Authentication is bilateral: at the end of the authentication exchange, each side is convinced of the other's identity. Every machine begins the exchange with a DES key in memory. In the case of CPU and file servers, the key, user name, and domain name for the server are read from permanent storage, usually non-volatile RAM. In the case of terminals, the key is derived from a password typed by the user at boot time. A special machine, known as the *authentication server*, maintains a database of keys for all users in its administrative domain and participates in the authentication protocols.

The authentication protocol is as follows: after exchanging challenges, one party contacts the authentication server to create permission-granting *tickets* encrypted with each party's secret key and containing a new conversation key. Each party decrypts its own ticket and uses the conversation key to encrypt the other party's challenge.

This structure is somewhat like Kerberos [MBSS87], but avoids its reliance on synchronized clocks. Also unlike Kerberos, Plan 9 authentication supports a 'speaks for' relation [LABW91] that enables one user to have the authority of another; this is how a CPU server runs processes on behalf of its clients.

Plan 9's authentication structure builds secure services rather than depending on firewalls. Whereas firewalls require special code for every service penetrating the wall, the Plan 9 approach permits authentication to be done in a single place—9P—for all services. For example, the `cpu` command works securely across the Internet.

Authenticating external connections

The regular Plan 9 authentication protocol is not suitable for text-based services such as Telnet or FTP. In such cases, Plan 9 users authenticate with hand-held DES calculators called *authenticators*. The authenticator holds a key for the user, distinct from the user's normal authentication key. The user 'logs on' to the authenticator using a 4-digit PIN. A correct PIN enables the authenticator for a challenge/response exchange with the server. Since a correct challenge/response exchange is valid only once and keys are never sent over the network, this procedure is not susceptible to replay attacks, yet is compatible with protocols like Telnet and FTP.

Special users

Plan 9 has no super-user. Each server is responsible for maintaining its own security, usually permitting access only from the console, which is protected by a password. For example, file servers have a unique administrative user called `adm`, with special privileges that apply only to commands typed at the server's physical console. These privileges concern the day-to-day maintenance of the server, such as adding new users and configuring disks and networks. The privileges do *not* include the ability to modify, examine, or change the permissions of any files. If a file is read-protected by a user, only that user may grant access to others.

CPU servers have an equivalent user name that allows administrative access to resources on that server such as the control files of user processes. Such permission is necessary, for example, to kill rogue processes, but does not extend beyond that server. On the other hand, by means of a key held in protected non-volatile RAM, the identity of the administrative user is proven to the authentication server. This allows the CPU server to authenticate remote users, both for access to the server itself and when the CPU server is acting as a proxy on their behalf.

Finally, a special user called `none` has no password and is always allowed to connect; anyone may claim to be `none`. `None` has restricted permissions; for example, it is not allowed to examine dump files and can read only world-readable files.

The idea behind `none` is analogous to the anonymous user in FTP services. On Plan 9, guest FTP servers are further confined within a special restricted name space. It disconnects guest users from system programs, such as the contents of `/bin`, but makes it possible to make local files available to guests by binding them explicitly into the space. A restricted name space is more secure than the usual technique of exporting an ad hoc directory tree; the result is a kind of cage around untrusted users.

The cpu command and proxied authentication

When a call is made to a CPU server for a user, say Peter, the intent is that Peter wishes to run processes with his own authority. To implement this property, the CPU server does the following when the call is received. First, the listener forks off a process to handle the call. This process changes to the user `none` to avoid giving away permissions if it is compromised. It then performs the authentication protocol to verify that the calling user really is Peter, and to prove to Peter that the machine is itself trustworthy. Finally, it reattaches to all relevant file servers using the authentication protocol to identify itself as Peter. In this case, the CPU server is a client of the file server and performs the client portion of the authentication exchange on behalf of Peter. The authentication server will give the process tickets to accomplish this only if the CPU server's administrative user name is allowed to *speak for* Peter.

The *speaks for* relation [LABW91] is kept in a table on the authentication server. To simplify the management of users computing in different authentication domains, it also contains mappings between user names in different domains, for example saying that user `rtm` in one domain is the same person as user `rtmorris` in another.

File Permissions

One of the advantages of constructing services as file systems is that the solutions to ownership and permission problems fall out naturally. As in UNIX, each file or directory has separate read, write, and execute/search permissions for the file's owner, the file's group, and anyone else. The idea of group is unusual: any user name is potentially a group name. A group is just a user with a list of other users in the group. Conventions make the distinction: most people have user names without group members, while groups have long lists of attached names. For example, the `sys` group traditionally has all the system programmers, and system files are accessible by group `sys`. Consider

the following two lines of a user database stored on a server:

```
pjw:pjw:
sys::pjw,ken,philw,presotto
```

The first establishes user `pjw` as a regular user. The second establishes user `sys` as a group and lists four users who are *members* of that group. The empty colon-separated field is space for a user to be named as the *group leader*. If a group has a leader, that user has special permissions for the group, such as freedom to change the group permissions of files in that group. If no leader is specified, each member of the group is considered equal, as if each were the leader. In our example, only `pjw` can add members to his group, but all of `sys`'s members are equal partners in that group.

Regular files are owned by the user that creates them. The group name is inherited from the directory holding the new file. Device files are treated specially: the kernel may arrange the ownership and permissions of a file appropriate to the user accessing the file.

A good example of the generality this offers is process files, which are owned and read-protected by the owner of the process. If the owner wants to let someone else access the memory of a process, for example to let the author of a program debug a broken image, the standard `chmod` command applied to the process files does the job.

Another unusual application of file permissions is the dump file system, which is not only served by the same file server as the original data, but represented by the same user database. Files in the dump are therefore given identical protection as files in the regular file system; if a file is owned by `pjw` and read-protected, once it is in the dump file system it is still owned by `pjw` and read-protected. Also, since the dump file system is immutable, the file cannot be changed; it is read-protected forever. Drawbacks are that if the file is readable but should have been read-protected, it is readable forever, and that user names are hard to re-use.

Performance

As a simple measure of the performance of the Plan 9 kernel, we compared the time to do some simple operations on Plan 9 and on SGI's IRIX Release 5.3 running on an SGI Challenge M with a 100MHz MIPS R4400 and a 1-megabyte secondary cache. The test program was written in Alef, compiled with the same compiler, and run on identical hardware, so the only variables are the operating system and libraries.

The program tests the time to do a context switch (`rendezvous` on Plan 9, `blockproc` on IRIX); a trivial system call (`rfork(0)` and `nap(0)`); and lightweight fork (`rfork(RFPROC)` and `sproc(PR_SFDS|PR_SADDR)`). It also measures the time to send a byte on a pipe from one process to another and the throughput on a pipe between two processes. The results appear in Table 1.

Test	Plan 9	IRIX
Context switch	39 µs	150 µs
System call	6 µs	36 µs
Light fork	1300 µs	2200 µs
Pipe latency	110 µs	200 µs
Pipe bandwidth	11678 KB/s	14545 KB/s

Table 1. Performance comparison.

Although the Plan 9 times are not spectacular, they show that the kernel is competitive with commercial systems.

Discussion

Plan 9 has a relatively conventional kernel; the system's novelty lies in the pieces outside the kernel and the way they interact. When building Plan 9, we considered all aspects of the system together, solving problems where the solution fit best. Sometimes the solution spanned many components. An example is the problem of heterogeneous instruction architectures, which is addressed by the compilers (different code characters, portable object code), the environment (`$cputype` and `$objtype`), the name space (binding in `/bin`), and other components. Sometimes many issues could be solved in a single place. The best example is 9P, which centralizes naming, access, and authentication. 9P is really the core of the system; it is fair to say that the Plan 9 kernel is primarily a 9P multiplexer.

Plan 9's focus on files and naming is central to its expressiveness. Particularly in distributed computing, the way things are named has profound influence on the system [Nee89]. The combination of local name spaces and global conventions to interconnect networked resources avoids the difficulty of maintaining a global uniform name space, while naming everything like a file makes the system easy to understand, even for novices. Consider the dump file system, which is trivial to use for anyone familiar with hierarchical file systems. At a deeper level, building all the resources above a single uniform interface makes interoperability easy. Once a resource exports a 9P interface, it can combine transparently with any other part of the system to build unusual applications; the details are hidden. This may sound object-oriented, but there are distinctions. First, 9P defines a fixed set of 'methods'; it is not an extensible protocol. More important, files are well-defined and well-understood and come prepackaged with familiar methods of access, protection, naming, and networking. Objects, despite their generality, do not come with these attributes defined. By reducing 'object' to 'file', Plan 9 gets some technology for free.

Nonetheless, it is possible to push the idea of file-based computing too far. Converting every resource in the system into a file system is a kind of metaphor, and metaphors can be abused. A good example of restraint is `/proc`, which is only a view of a process, not a representation. To run processes, the usual `fork` and `exec` calls are still necessary, rather than doing something like

```
cp /bin/date /proc/clone/mem
```

The problem with such examples is that they require the server to do things not under its control. The ability to assign meaning to a command like this does not imply the meaning will fall naturally out of the structure of answering the 9P requests it generates. As a related example, Plan 9 does not put machine's network names in the file name space. The network interfaces provide a very different model of naming, because using `open`, `create`, `read`, and `write` on such files would not offer a suitable place to encode all the details of call setup for an arbitrary network. This does not mean that the network interface cannot be file-like, just that it must have a more tightly defined structure.

What would we do differently next time? Some elements of the implementation are unsatisfactory. Using streams to implement network interfaces in the kernel allows protocols to be connected together dynamically, such as to attach the same TTY driver to TCP, URP, and IL connections, but Plan 9 makes no use of this configurability. (It was exploited, however, in the research UNIX system for which streams were invented.) Replacing streams by static I/O queues would simplify the code and make it faster.

Although the main Plan 9 kernel is portable across many machines, the file server is implemented separately. This has caused several problems: drivers that must be written twice, bugs that must be fixed twice, and weaker portability of the file system code. The solution is easy: the file server kernel should be maintained as a variant of the regular operating system, with no user processes and special compiled-in kernel processes

to implement file service. Another improvement to the file system would be a change of internal structure. The WORM jukebox is the least reliable piece of the hardware, but because it holds the metadata of the file system, it must be present in order to serve files. The system could be restructured so the WORM is a backup device only, with the file system proper residing on magnetic disks. This would require no change to the external interface.

Although Plan 9 has per-process name spaces, it has no mechanism to give the description of a process's name space to another process except by direct inheritance. The `cpu` command, for example, cannot in general reproduce the terminal's name space; it can only re-interpret the user's login profile and make substitutions for things like the name of the binary directory to load. This misses any local modifications made before running `cpu`. It should instead be possible to capture the terminal's name space and transmit its description to a remote process.

Despite these problems, Plan 9 works well. It has matured into the system that supports our research, rather than being the subject of the research itself. Experimental new work includes developing interfaces to faster networks, file caching in the client kernel, encapsulating and exporting name spaces, and the ability to re-establish the client state after a server crash. Attention is now focusing on using the system to build distributed applications.

One reason for Plan 9's success is that we use it for our daily work, not just as a research tool. Active use forces us to address shortcomings as they arise and to adapt the system to solve our problems. Through this process, Plan 9 has become a comfortable, productive programming environment, as well as a vehicle for further systems research.

References

[9man] *Plan 9 Programmer's Manual, Volume 1,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[ANSIC] *American National Standard for Information Systems – Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[Duff90] Tom Duff, ''Rc – A Shell for Plan 9 and UNIX systems'', *Proc. of the Summer 1990 UKUUG Conf.,* London, July, 1990, pp. 21–33, reprinted, in a different form, in this volume.

[Fra80] A.G. Fraser, ''Datakit – A Modular Network for Synchronous and Asynchronous Traffic'', *Proc. Int. Conf. on Commun.,* June 1980, Boston, MA.

[FSSUTF] *File System Safe UCS Transformation Format (FSS–UTF),* X/Open Preliminary Specification, 1993. ISO designation is ISO/IEC JTC1/SC2/WG2 N 1036, dated 1994–08–01.

[ISO10646] ISO/IEC DIS 10646–1:1993 *Information technology – Universal Multiple–Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

[Kill84] T.J. Killian, ''Processes as Files'', *USENIX Summer 1984 Conf. Proc.,* June 1984, Salt Lake City, UT.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, ''Authentication in Distributed Systems: Theory and Practice'', *Proc. 13th ACM Symp. on Op. Sys. Princ.,* Asilomar, 1991, pp. 165–182.

[MBSS87] S. P. Miller, B. C. Neumann, J. I. Schiller, and J. H. Saltzer, ''Kerberos Authentication and Authorization System'', Massachusetts Institute of Technology, 1987.

[NBS77] National Bureau of Standards (U.S.), *Federal Information Processing Standard 46,* National Technical Information Service, Springfield, VA, 1977.

[Nee89] R. Needham, ''Names'', in *Distributed systems,* S. Mullender, ed., Addison Wesley, 1989

[NeHe82] R.M. Needham and A.J. Herbert, *The Cambridge Distributed Computing System,* Addison–Wesley, London, 1982

[Neu92] B. Clifford Neuman, ''The Prospero File System'', *USENIX File Systems Workshop Proc.,* Ann Arbor, 1992, pp. 13–28.

[OCDNW88] John Ousterhout, Andrew Cherenson, Fred Douglis, Mike Nelson, and Brent Welch, ''The Sprite Network Operating System'', *IEEE Computer,* 21(2), 23–38, Feb. 1988.

[Pike87] Rob Pike, ''The Text Editor sam'', *Software – Practice and Experience,* Nov 1987, **17**(11), pp. 813–845; reprinted in this volume.

[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', *USENIX Summer Conf. Proc.,* Nashville, June, 1991, pp. 257–265, reprinted in this volume.

[Pike93] Rob Pike and Ken Thompson, ''Hello World or Καλημέρα κόσμε or こんにちは 世界'', *USENIX Winter Conf. Proc.,* San Diego, 1993, pp. 43–50, reprinted in this volume.

[Pike94] Rob Pike, ''Acme: A User Interface for Programmers'', *USENIX Proc. of the Winter 1994 Conf.,* San Francisco, CA,

[Pike95] Rob Pike, ''How to Use the Plan 9 C Compiler'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[POSIX] *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language],* IEEE, New York, 1990.

[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''The Use of Name Spaces in Plan 9'', *Op. Sys. Rev.,* Vol. 27, No. 2, April 1993, pp. 72–76, reprinted in this volume.

[Presotto]Dave Presotto, ''Multiprocessor Streams for Plan 9'', *UKUUG Summer 1990 Conf. Proc.,* July 1990, pp. 11–19.

[PrWi93] Dave Presotto and Phil Winterbottom, ''The Organization of Networks in Plan 9'', *USENIX Proc. of the Winter 1993 Conf.,* San Diego, CA, pp. 43–50, reprinted in this volume.

[PrWi95] Dave Presotto and Phil Winterbottom, ''The IL Protocol'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[RFC768] J. Postel, RFC768, *User Datagram Protocol, DARPA Internet Program Protocol Specification,* August 1980.

[RFC793] RFC793, *Transmission Control Protocol, DARPA Internet Program Protocol Specification,* September 1981.

[Rao91] Herman Chung-Hwa Rao, *The Jade File System,* (Ph. D. Dissertation), Dept. of Comp. Sci, University of Arizona, TR 91–18.

[Rit84] D.M. Ritchie, ''A Stream Input-Output System'', *AT&T Bell Laboratories Technical Journal,* **63**(8), October, 1984.

[Tric95] Howard Trickey, ''APE — The ANSI/POSIX Environment'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[Unicode]*The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1,* The Unicode Consortium, Addison Wesley, New York, 1991.

[UNIX85] *UNIX Time-Sharing System Programmer's Manual, Research Version, Eighth Edition, Volume 1.* AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[Welc94] Brent Welch, ''A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9'', *Computing Systems,* 7(2), pp. 175–199, Spring, 1994.

[Wint95] Phil Winterbottom, ''Alef Language Reference Manual'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression ‘‘the UNIX operating system.’’ The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on ‘‘the way things should be done.’’ Even so, if ‘‘the way’’ is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

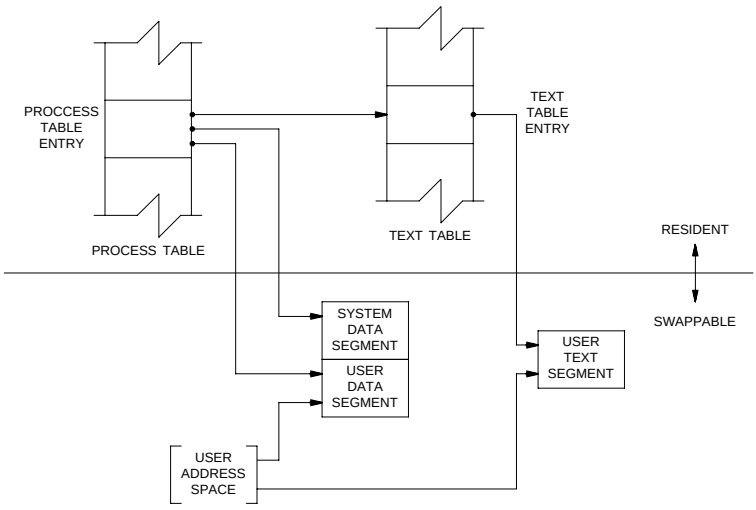


Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait**

for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a ''goto.'' If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a ''call.'' Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of ‘‘how much’’ can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been ‘‘structured I/O’’ and ‘‘unstructured I/O,’’ respectively; while the term ‘‘block I/O’’ has some meaning, ‘‘character I/O’’ is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in

most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as ''dirty.'' The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is ''cured'' by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the ''classical'' character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means ''everything other than block.'' I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue

will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or ‘‘borrow’’ block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a ‘‘disk’’ is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called ‘‘super-block.’’ This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases.

Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a ''triple indirect'' address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.,* 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

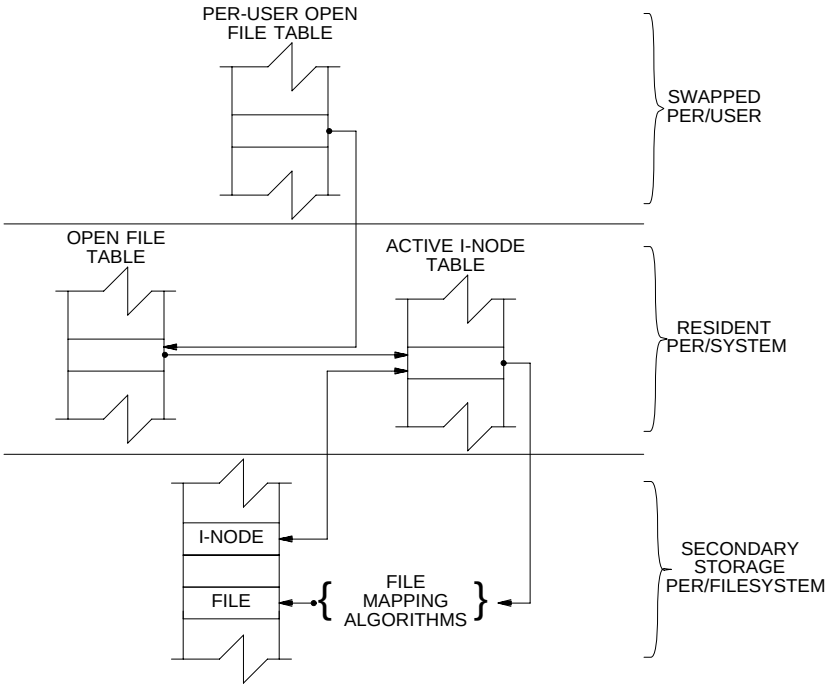


Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **fork**s) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipe**s consists of implied **seek**s before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user–a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing ''system'' code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, ''An Overview of SL5,'' *SIGPLAN Notices* **12**(4), pp.40-50 (April 1977).
2. E. W. Dijkstra, ''Cooperating Sequential Processes,'' pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).
3. J. A. Hawley and W. B. Meyer, ''MUNIX, A Multiprocessing Version of UNIX,'' M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' *Bell Sys. Tech. J.* **57**(6), pp.1905-1929 (1978).
5. S. R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell Sys. Tech. J.* **57**(6), pp.1971-1990 (1978).
6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11

CR Categories: 4.30, 4.32

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights. New York, October 15–17, 1973. Authors' address: Bell Laboratories, Murray Hill, NJ 07974.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [l] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator, macro processor (M6) [5], and permuted index program.

There is also a host of maintenance, utility, recreation, and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, UNIX documents are generated and formatted by the UNIX editor and text formatting program.

2. Hardware and Software Environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 144K bytes of core memory; UNIX occupies 42K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the

software mentioned above can require as little as 50K bytes of core altogether.

The PDP-11 has a 1M byte fixed-head disk, used for file system storage and swapping, four moving-head disk drives which each provide 2.5M bytes on removable disk cartridges, and a single moving-head disk drive which uses removable 40M byte disk packs. There are also a high-speed paper tape reader-punch, nine-track magnetic tape, and D-tape (a variety of magnetic tape facility in which individual records may be addressed and rewritten). Besides the console typewriter, there are 14 variable-speed communications interfaces attached to 100-series datasets and a 201 dataset interface used primarily for spooling printout to a communal line printer. There are also several one-of-a-kind devices including a Picturephone® interface, a voice response unit, a voice synthesizer, a phototypesetter, a digital switching network, and a satellite PDP-11/20 which generates vectors, curves, and characters on a Tektronix 611 storage-tube display.

The greater part of UNIX software is written in the above-mentioned C language [6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.

3. The File System

The most important job of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary Files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure: the assembler generates and the loader expects an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the *root* directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Another system directory contains all the programs provided for general use; that is, all the *commands*. As will be seen however, it is by no means necessary that a program reside in this directory for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes "/" and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name */alpha/beta/gamma* causes the system to search the root for directory *alpha*, then to search *alpha* for *beta*, finally to find *gamma* in *beta*. *Gamma* may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name *alpha/beta* specifies the file named *beta* in subdirectory *alpha* of the current directory. The simplest kind of name, for example *alpha*, refers to a file which itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same nondirectory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. UNIX differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other, which is its parent. The reason for this is to simplify the writing of programs which visit subtrees of the directory

structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special Files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev*, although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable File Systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a *mount* system request which has two arguments: the name of an existing ordinary file, and the name of a direct-access special file whose associated storage volume (e.g. disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of *mount* is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, *mount* replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the *mount*, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on the fixed-head disk, and the large disk drive, which contains user's files, is mounted by the system initialization program, the four smaller disk drives are available to users for mounting their own disk packs. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping which would otherwise be required to assure removal of the links when the removable volume is finally dismounted. In particular, in the root directories of all file systems, removable or not, the name ".." refers to the directory itself instead of to its parent.

3.5 Protection

Although the access control scheme in UNIX is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of seven protection bits. Six of these specify independently read, write, and execute permission for the owner of the file and for all other users.

If the seventh bit is on, the system will temporarily change the user identification of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program which calls for it. The set-user-ID feature provides for privileged programs which may use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by the program itself. If the set-user-identification bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands which call privileged system entries. For example, there is a system entry invocable only by the "super-user" (below) which creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Since anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available with- out administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed in [7].

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example) programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O Calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and sequential I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the

highest byte written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O in UNIX, Some of the basic calls are summarized below in an anonymous language which will indicate the required parameters without getting into the complexities of machine language programming. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

filep = open (name, flag)

Name indicates the name of the file. An arbitrary path name may be given. The *flag* argument indicates whether the file is to be read, written, or "updated", that is read and written simultaneously.

The returned value *filep* is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate it.

To create a new file or completely rewrite an old one, there is a *create* system call which creates the given file if it does not exist, or truncates it to zero length if it does exist. *Create* also opens the new file for writing and, like *open*, returns a file descriptor.

There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing; although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice, difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file which another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor which makes a copy of the file being edited.

It should be said that the system has sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in such inconvenient activities as writing on the same file, creating files in the same directory or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the first following byte. For each open file there is a pointer, maintained by the system, which indicates the next byte to be read or written. If n bytes are read or written, the pointer advances by n bytes.

Once a file is open, the following calls may be used:

n = read(filep, buffer, count)
n = write(filep, buffer, count)

Up to *count* bytes are transmitted between the file specified by *filep* and the byte array specified by *buffer*. The returned value *n* is the number of bytes actually transmitted. In the *write* case, *n* is the same as *count* except under exceptional conditions like I/O errors or end of physical medium on special files; in a read, however, *n* may without error be less than *count*. If the read pointer is so near the end of the file that reading count characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like devices never return more than one line of input. When a *read* call returns with *n* equal to zero, it indicates the end of the file. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a typewriter by use of an escape sequence which depends on the device used.

Bytes written on a file affect only those implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is grown as needed.

To do random (direct access) I/O, it is only necessary to move the read or write pointer to the appropriate location in the file.

location = seek(filep, base, offset)

The pointer associated with *filep* is moved to a position *offset* bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on *base*. *Offset* may be negative. For some devices (e.g. paper tape and typewriters) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in *location*.

3.6.1 Other I/O Calls. There are several additional system entries having to do with I/O and with the file system which will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

4. Implementation of the File System

As mentioned in §3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its *i-number* is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry thereby found (the file's *i-node*) contains the description of the file as follows.

1. Its owner.
2. Its protection bits.
3. The physical disk or tape addresses for the file contents.
4. Its size.

5. Time of last modification
6. The number of links to the file, that is, the number of times it appears in a directory.
7. A bit indicating whether the file is a directory.
8. A bit indicating whether the file is a special file.
9. A bit indicating whether the file is "large" or "small."

The purpose of an *open* or *create* system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the *open* or *create*. Thus the file descriptor supplied during a subsequent call to read or write the file may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made which contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is deallocated.

The space on all fixed or removable disks which contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit which depends on the device. There is space in the i-node of each file for eight device addresses. A *small* (nonspecial) file fits into eight or fewer blocks; in this case the addresses of the blocks themselves are stored. For *large* (nonspecial) files, each of the eight device addresses may point to an indirect block of 256 addresses of blocks constituting the file itself. These files may be as large as 8·256·512, or l,048,576 (2^{20}) bytes.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last seven device address words are immaterial, and the list is interpreted as a pair of bytes which constitute an internal *device* name. These bytes specify respectively a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar typewriter interfaces.

In this environment, the implementation of the *mount* system call (§3.4) is quite straightforward. *Mount* maintains a system table whose argument is the i-number and device name of the ordinary file specified during the *mount*, and whose corresponding value is the device name of the indicated special file. This table is searched for each (i-number, device)-pair which turns up while a path name is being scanned during an *open* or *create*; if a match is found, the i-number is replaced by 1 (which is the i-number of the root directory on all file systems), and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte.

UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer, and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

A program which reads or writes files in units of 512 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. A program which is used rarely or which does no great volume of I/O may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name which is related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, since it need only scan the linearly-organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, since all directory entries for a file have equal status. Charging the owner of a file is unfair, in general, since one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. The current version of UNIX avoids the issue by not charging any fees at all.

4.1 Efficiency of the File System

To provide an indication of the overall efficiency of UNIX and of the file system in particular, timings were made of the assembly of a 7621-line program. The assembly was run alone on the machine; the total clock time was 35.9 sec, for a rate of 212 lines per sec. The time was divided as follows: 63.5 percent assembler execution time, 16.5 percent system overhead, 20.0 percent disk wait time. We will not attempt any interpretation of these figures nor any comparison with other systems, but merely note that we are generally satisfied with the overall performance of the system.

5. Processes and Images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory, and the like. An image is the current state of a pseudo computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the fork system call:

processid = fork (label)

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label*. The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.

5.2 Pipes

Processes may communicate with related processes using the same system *read* and *write* calls that are used for file system I/O. The call

filep = pipe( )

returns a file descriptor *filep* and creates an interprocess channel called a *pipe*. This channel, like other open flies, is passed from parent to child process in the image by the *fork* call. A *read* using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although interprocess communication via pipes is a quite valuable tool (see §6.2), it is not a completely general mechanism since the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of Programs

Another major system primitive is invoked by

execute(file, arg_1, arg_2, ..., arg_n)

which requests the system to read in and execute the program named by *file*, passing it string arguments arg_1, arg_2, ..., arg_n. Ordinarily, arg_1 should be the same string as *file*, so that the program may determine the name by which it was invoked. All the code and data in the process using *execute* is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process Synchronization

Another process control system call

processid = wait( )

causes its caller to suspend execution until one of its children has completed execution. Then *wait* returns the *processid* of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. *Wait* may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination

Lastly,

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the *wait* primitive, the indicated *status* is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes

may also terminate as a result of various illegal actions or user-generated signals (§7 below).

6. The Shell

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 · · · arg_n

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the "/" character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string /*bin*/ to command and attempts again to find the file. Directory /*bin* contains all the commands intended to be generally used.

6.1 Standard I/O

The discussion of I/O in §3 above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the Shell, however, start off with two open files which have file descriptors 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's typewriter. Thus programs which wish to write informative or diagnostic information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs which wish to read messages typed by the user usually read this file.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. If one of the arguments 'to a command is prefixed by "⟩", file descriptor 1 will, for the duration of the command, refer to the file named after the "⟩". For example,

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command

ls ⟩there

creates a file called *there* and places the listing there. Thus the argument "⟩there" means, "place output on *there*." On the other hand,

ed

ordinarily enters the editor, which takes requests from the user via his typewriter. The command

ed ⟨script

interprets *script* as a file of editor commands; thus "⟨script" means, "take input from *script*."

Although the file name following "⟨" or "⟩" appears to be an argument to the command, in fact it is interpreted completely by the Shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line

ls | pr –2 | opr

ls lists the names of the files in the current directory; its output is passed to *pr*, which paginates its input with dated headings. The argument "–2" means double column. Likewise the output from *pr* is input to *opr*. This command spools its input onto a file for off-line printing.

This process could have been carried out more clumsily by

ls ⟩temp1
pr –2 ⟨temp1 ⟩temp2
opr ⟨temp2

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the *ls* command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as *ls* to provide such a wide variety of output options.

A program such as *pr* which copies its standard input to its standard output (with processing) is called a *filter*. Some filters which we have found useful perform character transliteration, sorting of the input, and encryption and decryption.

6.3 Command Separators: Multitasking

Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "&", the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example,

```
as source >output &
```

causes source to be assembled, with diagnostic output going to output; no matter how long the assembly takes, the Shell returns immediately. When the Shell does not wait for the completion of a command, the identification of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In the examples above using "&", an output file other than the typewriter was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The Shell also allows parentheses in the above operations. For example,

```
(date; ls) >x &
```

prints the current date and time followed by a list of the current directory onto the file *x*. The Shell also returns immediately for another request.

6.4 The Shell as a Command: Command files

The Shell is itself a command, and may be called recursively. Suppose file *tryout* contains the lines

```
as source
mv a.out testprog
testprog
```

The *mv* command causes the file *a.out* to be renamed *testprog*. *a.out* is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the console, *source* would be assembled, the resulting program named *testprog*, and *testprog* executed. When the lines are in *tryout*, the command

```
sh <tryout
```

would cause the Shell *sh* to execute the commands sequentially.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It is also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

6.5 Implementation of the Shell

The outline of the operation of the Shell can now be understood. Most of tile time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the Shell's *read* call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for *execute*. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an *execute* with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the *fork*, which is the parent process, *waits* for the child process to die. When this happens, the Shell knows the command is finished, so it types its prompt and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "&", the Shell merely refrains from waiting for the process which it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the *fork* primitive, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given however, the offspring process, just before it performs *execute*, makes the standard I/O file descriptor 0 or 1 respectively refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is *open*ed (or *create*d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the Shell need not know the actual names of the files which are its own standard input and output since it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the Shell never terminates. (The main loop includes that branch of the return from *fork* belonging to the parent process; that is, the branch which does a *wait*, then reads another command line.) The one thing which causes the Shell to terminate is discovering an end-of-file condition on its input file. Thus,

when the Shell is executed as a command with a given input file, as in

sh ⟨comfile

the commands in *comfile* will be executed until the end of *comfile* is reached; then the instance of the Shell invoked by *sh* will terminate. Since this Shell process is the child of another instance of the Shell, the *wait* executed in the latter will return, and another command may be processed.

6.6 Initialization

The instances of the Shell to which users type commands are themselves children of another process. The last step in the initialization of UNIX is the creation of a single process and the invocation (via *execute*) of a program called *init*. The role of *init* is to create one process for each typewriter channel which may be dialed up by a user. The various subinstances of *init* open the appropriate typewriters for input and output. Since when *init* was invoked there were no files open, in each process the typewriter keyboard will receive file descriptor 0 and the printer file descriptor 1. Each process types out a message requesting that the user log in and waits, reading the typewriter, for a reply. At the outset, no one is logged in, so each process simply hangs. Finally someone types his name or other identification. The appropriate instance of *init* wakes up, receives the log-in line, and reads a password file. If the user name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this path of *init* simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another login message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6.7 Other Programs as Shell

The Shell as described above is designed to allow users full access to the facilities of the system since it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

Recall that after a user has successfully logged in by supplying his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in tile password file may contain the name of a program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system specify that the editor *ed* is to be used instead of the Shell. Thus when editing system users log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking UNIX programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g. chess, blackjack, 3D tic-tac-toe) available on UNIX illustrate a much more severely restricted environment. For each of these an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the Shell. People who log in as a player of one of the games find themselves limited to the game and unable to investigate the presumably more interesting offerings of UNIX as a whole.

7. Traps

The PDP-11 hardware detects a number of program faults, such as references to nonexistent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. When an illegal action is caught, unless other arrangements have been made, the system terminates the process and writes the user's image on file *core* in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs which are looping, which produce unwanted output, or about which the user has second thoughts may be halted by the use of the *interrupt* signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a core image file.

There is also a *quit* signal which is used to force a core image to be produced. Thus programs which loop unexpectedly may be halted and the core image examined without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by the process. For example, the Shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating point hardware, unimplemented instructions are caught, and floating point instructions are interpreted.

8. Perspective

Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer

facilities, discovered a little-used system PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. Then in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations which influenced the design of UNIX are visible in retrospect.

First, since we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second there have always been fairly severe size constraints on the system and its software. Given the partiality antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third, nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long, which buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable from a space-efficiency standpoint to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load routines for dealing with each device with all programs, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process control scheme and command interface have proved both convenient and efficient. Since the Shell operates as an ordinary, swappable user program, it consumes no wired-down space in the system proper, and it may be made as powerful as desired at little cost, In particular, given the framework in which the Shell executes as a process which spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

8.1 Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The *fork* operation, essentially as we implemented it, was present in the Berkeley time-sharing system [8]. On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls [9] and both the name of the Shell and its general functions, The notion that the Shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX [10].

9. Statistics

The following statistics from UNIX are presented to show the scale of the system and to show how a system of this scale is used. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

9.1 Overall

72 user population
14 maximum simultaneous users
300 directories
4400 files
34000 512-byte secondary storage blocks used

9.2 Per day (24-hour day, 7-day week basis)

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant $e - 2$, and is now generating composite pseudoprimes (base 2).

1800 commands
4.3 CPU hours (aside from background)
70 connect hours
30 different users
75 logins

9.3 Command CPU Usage (cut off at 1%)

15.7%	C compiler	1.7%	Fortran compiler
15.2%	users' programs	1.6%	remove file
11.7%	editor	1.6%	tape archive
5.8%	Shell (used as a command, including command times)	1.6%	file system consistency check
		1.4%	library maintainer
5.3%	chess	1.3%	concatenate/print files
3.3%	list directory	1.3%	paginate and print file
3.1%	document formatter	1.1%	print disk usage
1.6%	backup dumper	1.0%	copy file
1.8%	assembler		

9.4 Command Accesses (cut off at 1%)

15.3%	editor	1.6%	debugger
9.6%	list directory	1.6%	Shell (used as a command)
6.3%	remove file	1.5%	print disk availability
6.3%	C compiler	1.4%	list processes executing
6.0%	concatenate/print file	1.4%	assembler
6.0%	users' programs	1.4%	print arguments
3.3%	list people logged on system	1.2%	copy file
		1.1%	paginate and print file
3.2%	rename/move file	1.1%	print current date/time
3.1%	file status	1.1%	file system consistency check
1.8%	library maintainer		
1.8%	document formatter	1.0%	tape archive
1.6%	execute another command conditionally		

9.5 Reliability

Our statistics on reliability are much more subjective than the others. The following results are true to the best of our combined recollections. The time span is over one year with a very early vintage 11/45.

There has been one loss of a file system (one disk out of five) caused by software inability to cope with a hard ware problem causing repeated power fail traps. Files on that disk were backed up three days.

A "crash" is an unscheduled system reboot or halt. There is about one crash every other day; about two-thirds of them are caused by hardware-related difficulties such as power dips and inexplicable processor interrupts to random locations. The remainder are software failures. The longest uninterrupted up time was about two weeks. Service calls average one every three weeks, but are heavily clustered. Total up time has been about 98 percent of our 24-hour, 365-day schedule.

Acknowledgments. We are grateful to R.H. Canaday, L.L. Cherry, and L.E. McMahon for their contributions to UNIX. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M.D. McIlroy, and J.F. Ossanna.

References

1. Digital Equipment Corporation. PDP-11/40 Processor Handbook, 1972, and PDP-11/45 Processor Handbook. 1971.
2. Deutsch, L.P., and Lampson, B.W. An online editor. *Comm. ACM 10*, 12 (Dec, 1967) 793–799, 803.
3. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557–566.
4. McClure, R.M. TMG—A syntax directed compiler. Proc. ACM 20th Nat. Conf., ACM, 1965, New York, pp. 262–274.
5. Hall. A.D. The M6 macroprocessor. Computing Science Tech. Rep. #2, Bell Telephone Laboratories, 1969.
6. Ritchie, D.M. C reference manual. Unpublished memorandum, Bell Telephone Laboratories, 1973.
7. Aleph-null. Computer Recreations. *Software Practice and Experience 1*, 2 (Apr.–June 1971), 201–204.
8. Deutsch, L.P., and Lampson, B.W. SDS 930 time-sharing system preliminary reference manual. Doc. 30.10.10, Project GENIE, U of California at Berkeley, Apr. 1965.
9. Feiertag. R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symp. on Oper. Syst. Princ., Oct. 18–20, 1971, ACM, New York, pp. 35–41.
10. Bobrow, D.C., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15*, 3 (Mar. 1972) 135–143.

Programming Techniques

R. M. McCLURE, Editor

Regular Expression Search Algorithm

KEN THOMPSON
Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

A method for locating specific character strings embedded in character text is described and an implementation of this method in the form of a compiler is discussed. The compiler accepts a regular expression as source language and produces an IBM 7094 program as object language. The object program then accepts the text to be searched as input and produces a signal every time an embedded string in the text matches the given regular expression. Examples, problems, and solutions are also presented.

KEY WORDS AND PHRASES: search, match, regular expression
CR CATEGORIES: 3.74, 4.49, 5.32

The Algorithm

Previous search algorithms involve backtracking when a partially successful search path fails. This necessitates a lot of storage and bookkeeping, and executes slowly. In the regular expression recognition technique described in this paper, each character in the text to be searched is examined in sequence against a list of all possible current characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the next character is obtained, and the process continues. In the terms of Brzozowski [1], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched. The parallel nature of this algorithm makes it extremely fast.

The Implementation

The specific implementation of this algorithm is a compiler that translates a regular expression into IBM 7094 code. The compiled code, along with certain runtime routines, accepts the text to be searched as input and finds all substrings in the text that match the regular expression. The compiling phase of the implemention does not detract from the overall speed since any search routine must translate the input regular expression into some sort of machine accessible form.

In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression.

This compile-search algorithm is incorporated as the context search in a time-sharing text editor. This is by no means the only use of such a search routine. For example, a variant of this algorithm is used as the symbol table search in an assembler.

It is assumed that the reader is familiar with regular expressions [2] and the machine language of the IBM 7094 computer [3].

The Compiler

The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "$\cdot$" for juxtaposition of regular expressions. The second stage converts the regular expression to reverse Polish form. The third stage is the object code producer. The first two stages are straightforward and are not discussed. The third stage expects a syntactically correct, reverse Polish regular expression.

The regular expression $a(b \mid c)*d$ will be carried through as an example. This expression is translated into $abc \mid * \cdot d \cdot$ by the first two stages. A functional description of the third stage of the compiler follows:

The heart of the third stage is a pushdown stack. Each entry in the pushdown stack is a pointer to the compiled code of an operand. When a binary operator ("|" or "$\cdot$") is compiled, the top (most recent) two entries on the stack are combined and a resultant pointer for the operation replaces the two stack entries. The result of the binary operator is then available as an operand in another operation. Similarly, a unary operator ("*") operates on the top entry of the stack and creates an operand to replace that entry. When the entire regular expression is compiled, there is just one entry in the stack, and that is a pointer to the code for the regular expression.

The compiled code invokes one of two functional routines. The first is called NNODE. NNODE matches a single character and will be represented by an oval containing the character that is recognized. The second functional routine is called CNODE. CNODE will split the

current search path. It is represented by ⊕ with one input path and two output paths.

Figure 1 shows the functions of the third stage of the compiler in translating the example regular expression. The first three characters of the example *a*, *b*, *c*, each create a stack entry, *S*[*i*], and an NNODE box.

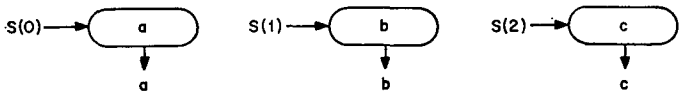


FIG. 1

The next character "*" combines the operands *b* and *c* with a CNODE to form *b*|*c* as an operand. (See Figure 2.)

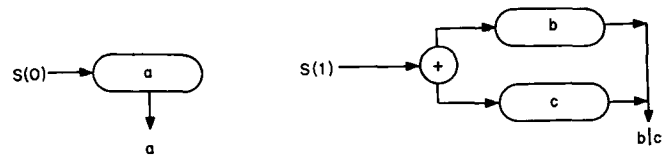


FIG. 2

The next character "*" operates on the top entry on the stack. The closure operator is realized with a CNODE by noting the identity $X* = \lambda|XX*$, where X is any regular expression (operand) and λ is the null regular expression. (See Figure 3.)

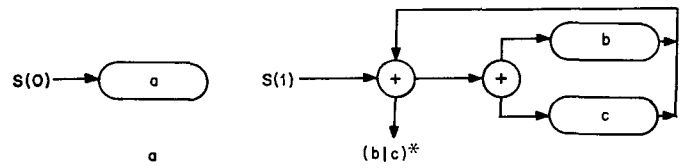


FIG. 3

The next character "·" compiles no code, but just combines the top two entries on the stack to be executed sequentially. The stack now points to the single operand $a \cdot (b|c)*$. (See Figure 4.)

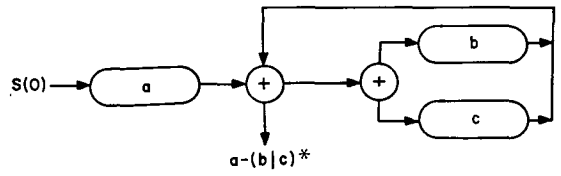


FIG. 4

The final two characters *d*· compile and connect an NNODE onto the existing code to produce the final regular expression in the only stack entry. (See Figure 5.)

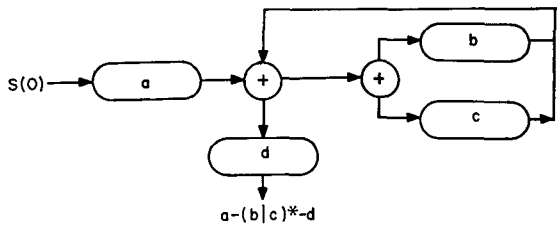


FIG. 5

A working example of the third stage of the compiler appears below. It is written in ALGOL-60 and produces object programs in IBM 7094 machine language.

```
begin
  integer procedure get character; code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol); code;
  integer procedure index(character); code;
  integer char, lc, pc;
  integer array stack[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  pc := pc+4;
  lc := lc+1;
  go to advance;
juxta:
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[stack[lc-1]] := instruction('tra', value('code')+pc, 0, 0);
  pc := pc+2;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The integer procedure *get character* returns the next character from the second stage of the compiler. The

integer procedure *index* returns an integer index to classify the character. The integer procedure *value* returns the location of a named subroutine. It is an assembler symbol table routine. The integer procedure *instruction* returns an assembled 7094 instruction.

When the compiler receives the example regular expression, the following 7094 code is produced:

```
CODE    TRA     CODE+1          0       a
        TXL     FAIL,1,-'a'-1   1
        TXH     FAIL,1,-'a'     2
        TSX     NNODE,4         3
        TRA     CODE+16         4       b
        TXL     FAIL,1,-'b'-1   5
        TXH     FAIL,1,-'b'     6
        TSX     NNODE,4         7
        TRA     CODE+16         8       c
        TXL     FAIL,1,-'c'-1   9
        TXH     FAIL,1,-'c'     10
        TSX     NNODE,4         11
        TRA     CODE+16         12      |
        TSX     CNODE,4         13
        TRA     CODE+9          14
        TRA     CODE+5          15
        TSX     CNODE,4         16      *
        TRA     CODE+13         17
        TRA     CODE+19         18      ·d
        TXL     FAIL,1,-'d'-1   19
        TXH     FAIL,1,-'d'     20
        TSX     NNODE,4         21
        TRA     FOUND           22      ·eof
```

Runtime Routines

During execution of the code produced by the compiler, two lists (named CLIST and NLIST) are maintained by the subroutines CNODE and NNODE. CLIST contains a list of TSX **,2 instructions terminated by a TRA XCHG. Each TSX represents a partial match of the regular expression and the TRA XCHG represents the end of the list of possible matches. A call to CNODE from location x moves the TRA XCHG instruction down one location in CLIST and inserts in its place a TSX $x+1,2$ instruction. Control is then returned to $x+2$. This effectively branches the current search path. The path at $x+1$ is deferred until later while the branch at $x+2$ is searched immediately. The code for CNODE is as follows:

```
CNODE   AXC   **,7          CLIST COUNT
        CAL   CLIST,7
        SLW   CLIST+1,7     MOVE TRA XCHG
                              INSTRUCTION
        PCA   ,4
        ACL   TSXCMD
        SLW   CLIST,7       INSERT NEW TSX **,2
                              INSTRUCTION
        TXI   *+1,7,-1
        SCA   CNODE,7       INCREMENT CLIST
                              COUNT
        TRA   2,4           RETURN
*
TSXCMD  TSX   1,2           CONSTANT, NOT
                              EXECUTED
```

The subroutine NNODE is called after a successful match of the current character. This routine, when called from location x, places a TSX $x+1,2$ in NLIST. It then returns to the next instruction in CLIST. This sets up the place in CODE to be executed with the next character. The code for NNODE is as follows:

```
NNODE   AXC   **,7          NLIST COUNT
        PCA   ,4
        ACL   TSXCMD
        SLW   NLIST,7       PLACE NEW TSX **,2
                              INSTRUCTION
        TXI   *+1,7,-1
        SCA   NNODE,7       INCREMENT NLIST
                              COUNT
        TRA   1,2
```

The routine FAIL simply returns to the next entry in the current list CLIST.

```
FAIL    TRA   1,2
```

The routine XCHG is transferred to when the current list is exhausted. This routine copies NLIST onto CLIST, appends a TRA XCHG instruction, gets a new character in index register one, and transfers to CLIST. The instruction TSX CODE,2 is also executed to start a new search of the entire regular expression with each character. Thus the regular expression will be found anywhere in the text to be searched. Variations can be easily incorporated. The code for XCHG is:

```
XCHG    LAC   NNODE,7       PICK UP NLIST COUNT
        AXC   0,6           PICK UP CLIST COUNT
X1      TXL   X2,7,0
        TXI   *+1,7,1
        CAL   NLIST,7
        SLW   CLIST,6       COPY NLIST ONTO CLIST
        TXI   X1,6,-1
X2      CLA   TRACMD
        SLW   CLIST,6       PUT TRA XCHG AT
                              BOTTOM
        SCA   CNODE,6       INITIALIZE CNODE
                              COUNT
        SCA   NNODE,0       INITIALIZE NNODE
                              COUNT
        TSX   GETCHA,4
        PAC   ,1            GET NEXT CHARACTER
        TSX   CODE,2        START SEARCH
        TRA   CLIST         FINISH SEARCH
*
TRACMD  TRA   XCHG          CONSTANT, NOT
                              EXECUTED
```

Initialization is required to set up the initial lists and start the first character.

```
INIT    SCA   NNODE,0
        TRA   XCHG
```

The routine FOUND is transferred to for each successful match of the entire regular expression. There is a one character delay between the end of a successful match and the transfer to FOUND. The null regular expression is found on the first character while one character regular expressions are found on the second character. This means that an extra (end of file) character must be put through

the code in order to obtain complete results. FOUND depends upon the use of the search routine and is therefore not discussed in detail.

The integer procedure GETCHA (called from XCHG) obtains the next character from the text to be searched. This character is right adjusted in the accumulator. GETCHA must also recognize the end of the text and terminate the search.

Notes

Code compiled for $a**$ will go into a loop due to the closure operator on an operand containing the null regular expression, λ. There are two ways out of this problem. The first is to not allow such an expression to get through the syntax sieve. In most practical applications, this would not be a serious restriction. The second way out is to recognize lambda separately in operands and remember the CODE location of the recognition of lambda. This means that $a*$ is compiled as a search for $\lambda|aa*$. If the closure operation is performed on an operand containing lambda, the instruction TRA FAIL is overlaid on that portion of the operand that recognizes lambda. Thus $a**$ is compiled as $\lambda|aa*(aa*)*$.

The array *lambda* is added to the third stage of the previous compiler. It contains zero if the corresponding operand does not contain λ. It contains the *code* location of the recognition of λ if the operand does contain λ. (The *code* location of the recognition of λ can never be zero.)

```
begin
  integer procedure get character; code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol); code;
  integer procedure index(character); code;
  integer char, lc, pc;
  integer array stack, lambda[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  lambda[lc] := 0;
  pc := pc+4;
  lc := lc+1;
 go to advance;
juxta:
  if lambda[lc-1] = 0 then
    lambda[lc-2] := 0;
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[pc+2] := instruction('tra', value('code')+pc+6, 0, 0);
  code[pc+3] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+4] := code[stack[lc-1]];
  code[pc+5] := instruction('tra', value('code')+pc+6, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+3, 0, 0);
  if lambda[lc-1] ≠ 0 then
    code[lambda[lc-1]] := instruction('tra', value('fail'), 0, 0);
  lambda[lc-1] := pc+5;
  pc := pc+6;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0)
  if lambda[lc-2] = 0 then
    begin if lambda[lc-1] ≠ 0 then
      lambda[lc-2] = lambda[lc-1]
    end else
    if lambda[lc-1] ≠ 0 then
      code[lambda[lc-1]] :=
        instruction('tra', value('code')+lambda[lc-2], 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The next note on the implementation is that the sizes of the two runtime lists can grow quite large. For example, the expression $a*a*a*a*a*a*a*$ explodes when it encounters a few concurrent a's. This expression is equivalent to $a*$ and therefore should not generate so many entries. Such redundant searches can be easily terminated by having NNODE (CNODE) search NLIST (CLIST) for a matching entry before it puts an entry in the list. This now gives a maximum size on the number of entries that can be in the lists. The maximum number of entries that can be in CLIST is the number of TSX CNODE,4 and TSX NNODE,4 instructions compiled. The maximum number of entries in NLIST is just the number of TSX NNODE,4 instructions compiled. In practice, these maxima are never met.

The execution is so fast, that any other recognition and deleting of redundant searches, such as described by Kuno and Oettinger [4], would probably waste time.

This compiling scheme is very amenable to the extension of the regular expressions recognized. Special characters can be introduced to match special situations or sequences. Examples include: beginning of line character, end of line character, any character, alphabetic character, any number of spaces character, lambda, etc. It is also easy to incorporate new operators in the regular expression routine. Examples include: not, exclusive or, intersection, etc.

REFERENCES

1. Brzozowski, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct. 1964), 481–494.
2. Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, Ann. Math. Stud. No. 34. Princeton U. Press, Princeton, N.J., 1956, pp. 3–41.
3. IBM Corp. IBM 7094 principles of operation. File No. 7094-01, Form A22-6703-1.
4. Kuno, S., and Oettinger, A. G. Multiple-path syntactic analyzer. Proc. IFIP Congress, Munich, 1962, North-Holland Pub. Co., Amsterdam.

Hello World
or
Καλημέρα κόσμε
or
こんにちは 世界

Rob Pike
Ken Thompson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plan 9 from Bell Labs has recently been converted from ASCII to an ASCII-compatible variant of Unicode, a 16-bit character set. In this paper we explain the reasons for the change, describe the character set and representation we chose, and present the programming models and software changes that support the new text format. Although we stopped short of full internationalization—for example, system error messages are in Unixese, not Japanese—we believe Plan 9 is the first system to treat the representation of all major languages on a uniform, equal footing throughout all its software.

Introduction

The world is multilingual but most computer systems are based on English and ASCII. The release of Plan 9 [Pike90], a new distributed operating system from Bell Laboratories, seemed a good occasion to correct this chauvinism. It is easier to make such deep changes when building new systems than by refitting old ones.

The ANSI C standard [ANSIC] contains some guidance on the matter of 'wide' and 'multi-byte' characters but falls far short of solving the myriad associated problems. We could find no literature on how to convert a *system* to larger character sets, although some individual *programs* had been converted. This paper reports what we discovered as we explored the problem of representing multilingual text at all levels of an operating system, from the file system and kernel through the applications and up to the window system and display.

Plan 9 has not been 'internationalized': its manuals are in English, its error messages are in English, and it can display text that goes from left to right only. But before we can address these other problems, we need to handle, uniformly and comfortably, the textual representation of all the major written languages. That subproblem is richer than we had anticipated.

Standards

Our first step was to select a standard. At the time (January 1992), there were only two viable options: ISO 10646 [ISO10646] and Unicode [Unicode]. The documents describing both proposals were still in the draft stage.

ISO 10646 was not very attractive to us. The standard defines a sparse set of 32-bit characters, which would be hard to implement and have punitive storage requirements. Also, the standard attempts to mollify national interests by allocating 16-bit subspaces to national committees to partition individually. The suggested mode of use is to "flip" between separate national standards to implement the international standard. This did not strike us as a sound basis for a character set. As well, transmitting 32-bit values in a

byte stream, such as in pipes, would be expensive and hard to implement. Since the standard does not define a byte order for such transmission, the byte stream would also have to carry state to enable the values to be recovered.

Unicode is a proposal by a consortium of mostly American computer companies formed to protest the technical failings of ISO 10646. Unicode defines a uniform 16-bit code based on the principle of unification: two characters are the same if they look the same even though they are from different languages. This principle, called Han unification, allows the large Japanese, Chinese, and Korean character sets to be packed comfortably into a 16-bit representation.

We chose Unicode for its technical merits and because its code space was better defined. Moreover, the existence of Unicode was derailing the ISO 10646 standard. ISO 10646 is now in its second draft and has only one 16-bit group defined, which is almost exactly Unicode. Most people expect the two standards bodies to reach a détente so that ISO 10646 and Unicode will represent the same character set.

Unicode defines an adequate character set but an unreasonable representation. The Unicode standard states that all characters are 16 bits wide and are communicated and stored in 16-bit units. It also reserves a pair of characters (hexadecimal FFFE and FEFF) to detect byte order in transmitted text, requiring state in the byte stream. (The Unicode committee was thinking of files, not pipes.) To adopt Unicode, we would have had to convert all text going into and out of Plan 9 between ASCII and Unicode, which cannot be done. Within a single program, in command of all its input and output, it is possible to define characters as 16-bit quantities; in the context of a networked system with hundreds of applications on diverse machines by different manufacturers, it is impossible.

We needed a way to adapt Unicode to the tools-and-pipes model of text processing embodied by the Unix system. To do that, we needed an ASCII-compatible textual representation of Unicode for transmission and storage. In the ISO standard there is an informative (non-required) Annex called UTF that provides a byte stream encoding of the 32-bit ISO code. The encoding uses multibyte sequences composed from the 190 printable characters of Latin-1 to represent character values larger than 159.

The UTF encoding has several good properties. By far the most important is that a byte in the ASCII range 0-127 represents itself in UTF. Thus UTF is backward compatible with ASCII.

UTF has other advantages. It is a byte encoding and is therefore byte-order independent. ASCII control characters appear in the byte stream only as themselves, never as an element of a sequence encoding another character, so newline bytes separate lines of UTF text. Finally, ANSI C's `strcmp` function applied to UTF strings preserves the ordering of Unicode characters.

To encode and decode UTF is expensive (involving multiplication, division, and modulo operations) but workable. UTF's major disadvantage is that the encoding is not self-synchronizing. It is in general impossible to find the character boundaries in a UTF string without reading from the beginning of the string, although in practice control characters such as newlines, tabs, and blanks provide synchronization points.

In August 1992, X-Open circulated a proposal for another UTF-like byte encoding of Unicode. Their major concern was that an embedded character in a file name (in particular a slash) could be part of an escape sequence in UTF and therefore confuse a traditional file system. Their proposal would allow all 7-bit ASCII characters to represent themselves *and only themselves* in text. Multibyte sequences would contain only characters with the high bit set. We proposed a modification to the new UTF that would address our synchronization problem. The modified new proposal is now informally called UTF-2 and is being proposed as another informative Annex to ISO 10646.

The model for text in Plan 9 is chosen from these three standards*: the Unicode character set encoded as byte stream by UTF-2, from an X-Open proposed modification of Annex F of ISO 10646. Although this may seem like a precarious position for us to adopt, it is not as bad as it sounds. If, as expected, ISO adopts Unicode as Group 0 of 10646 and ISO publishes UTF-2 as an Annex, then Plan 9 will be ISO/UTF-2 compatible.

There are a couple of aspects of Unicode we have not faced. One is the issue of right-to-left text such

* "That's the nice thing about standards—there's so many to choose from." – Andy Tannenbaum (no, the other one)

as Hebrew or Arabic. Since that is an issue of display, not representation, we believe we can defer that problem for the moment without affecting our ability to solve it later. Another issue is diacriticals, which cause overstriking of multiple Unicode characters. Again, these are display issues and, since the Unicode committee is still deciding their finer points, we felt comfortable deferring. Mañana.

Although we converted Plan 9 in the altruistic interests of serving foreign languages, we have found the large character set attractive for other reasons. Unicode includes many characters—mathematical symbols, scientific notation, more general punctuation, and more—that we now use daily in our work. We no longer test our imaginations to find ways to include non-ASCII symbols in our text; why type `:-)` when you can use the character ☺? Most compelling is the ability to absorb documents and data that contain non-ASCII characters; our browser for the Oxford English Dictionary lets us see the dictionary as it really is, with pronunciation in the IPA font, foreign phrases properly rendered, and so on, *in plain text.*

In the rest of this paper, except when stated otherwise, the term 'UTF' refers to the UTF-2 encoding of Unicode characters as adopted by Plan 9.

C Compiler

The first program to be converted to UTF was the C Compiler. There are two levels of conversion. On the syntactic level, input to the C compiler is UTF; on the semantic level, the C language needs to define how compiled programs manipulate the UTF set.

The syntactic part is simple. The ANSI C language standard defines the source character set to be ASCII. Since UTF is backward compatible with ASCII, the compiler needs little change. The only places where a larger character set is allowed are in character constants, strings, and comments. Since 7-bit ASCII characters can represent only themselves in UTF, the compiler does not have to be careful while looking for the termination of a string or comment.

The Plan 9 compiler extends ANSI C to treat any Unicode character with a value outside of the ASCII range as an alphabetic. To a Greek programmer or an English mathematician, α is a sensible and now valid variable name.

On the semantic level, ANSI C allows, but does not tie down, the notion of a *wide character* and admits string and character constants of this type. We chose the wide character type to be `unsigned short`. In the libraries, the word `Rune` is defined by a `typedef` to be equivalent to `unsigned short` and is used to signify a Unicode character.

There are surprises; for example:

`L'x'`	is 120
`'x'`	is 120
`L'ÿ'`	is 255
`'ÿ'`	is −1, stdio `EOF` (if `char` is signed)
`L'α'`	is 945
`'α'`	is illegal

In the string constants,

```
"こんにちは 世界"
L"こんにちは 世界",
```

the former is an array of `chars` with 22 elements and a null byte, while the latter is an array of `unsigned shorts` (`Runes`) with 8 elements and a null `Rune`.

The Plan 9 library provides an output conversion function, `print` (analogous to `printf`), with formats `%c`, `%C`, `%s`, and `%S`. Since `print` produces text, its output is always UTF. The character conversion `%c` (lower case) masks its argument to 8 bits before converting to UTF. Thus `L'ÿ'` and `'ÿ'` printed under `%c` will be identical, but `L'α'` will print as the Unicode character with decimal value 177. The character conversion `%C` (upper case) masks its argument to 16 bits before converting to UTF. Thus `L'ÿ'` and `L'α'` will print correctly under `%C`, but `'ÿ'` will not. The conversion `%s` (lower case) expects a pointer to `char` and copies UTF sequences up to a null byte. The conversion `%S` (upper case) expects a pointer to `Rune` and performs sequential `%C` conversions until a null `Rune` is encountered.

Another problem in format conversion is the definition of `%10s`: does the number refer to bytes or characters? We decided that such formats were most often used to align output columns and so made the number count characters. Some programs, however, use the count to place blank-padded strings in fixed-sized arrays. These programs must be found and corrected.

Here is a complete example:

```
#include <u.h>

char c[] = "こんにちは 世界";
Rune s[] = L"こんにちは 世界";

main(void)
{
        print("%d, %d\n", sizeof(c), sizeof(s));
        print("%s\n", c);
        print("%S\n", s);
}
```

This program prints `23, 18` and then two identical lines of UTF text. In practice, `%S` and `L"..."` are rare in programs; one reason is that most formatted I/O is done in unconverted UTF.

Ramifications

All programs in Plan 9 now read and write text as UTF, not ASCII. This change breaks two deep-rooted symmetries implicit in most C programs:

1. A character is no longer a `char`.
2. The internal representation (Unicode) of a character now differs from its external representation (UTF).

In the sections that follow, we show how these issues were faced in the layers of system software from the operating system up to the applications. The effects are wide-reaching and often surprising.

Operating system

Since UTF is the only format for text in Plan 9, the interface to the operating system had to be converted to UTF. Text strings cross the interface in several places: command arguments, file names, user names (people can log in using their native name), error messages, and miscellaneous minor places such as commands to the I/O system. Little change was required: null-terminated UTF strings are equivalent to null-terminated ASCII strings for most purposes of the operating system. The library routines described in the next section made that change straightforward.

The window system, once called `8.5`, is now rightfully called 8½.

Libraries

A header file included by all programs (see [Pike92]) declares the `Rune` type to hold 16-bit character values:

```
typedef unsigned short Rune;
```

Also defined are several constants relevant to UTF:

```
enum
{
    UTFmax    = 3,     /* maximum bytes per rune */
    Runesync  = 0x80,  /* cannot represent part of a UTF sequence (<) */
    Runeself  = 0x80,  /* rune and UTF sequences are the same (<) */
    Runeerror = 0x80,  /* decoding error in UTF */
};
```

(With the original UTF, `Runesync` was hexadecimal 21 and `Runeself` was A0.) `UTFmax` bytes are

sufficient to hold the UTF encoding of any Unicode character. Characters of value less than `Runesync` only appear in a UTF string as themselves, never as part of a sequence encoding another character. Characters of value less than `Runeself` encode into single bytes of the same value. Finally, when the library detects errors in UTF input—byte sequences that are not valid UTF sequences—it converts the first byte of the error sequence to the character `Runeerror`. There is little a rune-oriented program can do when given bad data except exit, which is unreasonable, or carry on. Originally the conversion routines, described below, returned errors when given invalid UTF, but we found ourselves repeatedly checking for errors and ignoring them. We therefore decided to convert a bad sequence to a valid rune and continue processing. (The ANSI C routines, on the other hand, return errors.)

This technique does have the unfortunate property that converting invalid UTF byte strings in and out of runes does not preserve the input, but this circumstance only occurs when non-textual input is given to a textual program. Unicode defines an error character, value FFFD, to represent characters from other sets that are not represented in Unicode. The `Runeerror` character is a different concept, related to UTF rather than Unicode, so we chose a different character for it.

The Plan 9 C library contains a number of routines for manipulating runes. The first set converts between runes and UTF strings:

```
extern  int     runetochar(char*, Rune*);
extern  int     chartorune(Rune*, char*);
extern  int     runelen(long);
extern  int     fullrune(char*, int);
```

`Runetochar` translates a single `Rune` to a UTF sequence and returns the number of bytes produced. `Chartorune` goes the other way, reporting how many bytes were consumed. `Runelen` returns the number of bytes in the UTF encoding of a rune. `Fullrune` examines a UTF string up to a specified number of bytes and reports whether the string begins with a complete UTF encoding. All these routines use the `Runeerror` character to work around encoding problems.

There is also a set of routines for examining null-terminated UTF strings, based on the model of the ANSI standard `str` routines, but with `utf` substituted for `str` and `rune` for `chr`:

```
extern  int     utflen(char*);
extern  char*   utfrune(char*, long);
extern  char*   utfrrune(char*, long);
extern  char*   utfutf(char*, char*);
```

`Utflen` returns the number of runes in a UTF string; `utfrune` returns a pointer to the first occurrence of a rune in a UTF string; and `utfrrune` a pointer to the last. `Utfutf` searches for the first occurrence of a UTF string in another UTF string. Given the synchronizing property of UTF-2, `utfutf` is the same as `strstr` if the arguments point to valid UTF strings.

It is a mistake to use `strchr` or `strrchr` unless searching for a 7-bit ASCII character, that is, a character less than `Runeself`.

We have no routines for manipulating null-terminated arrays of `Runes`. Although they should probably exist for completeness, we have found no need for them, for the same reason that `%S` and `L"..."` are rarely used.

Most Plan 9 programs use a new buffered I/O library, BIO, in place of Standard I/O. BIO contains routines to read and write UTF streams, converting to and from runes. `Bgetrune` returns, as a `Rune` within a `long`, the next character in the UTF input stream; `Bputrune` takes a rune and writes its UTF representation. `Bungetrune` puts a rune back into the input stream for rereading.

Plan 9 programs use a simple set of macros to process command line arguments. Converting these macros to UTF automatically updated the argument processing of most programs. In general, argument flag names can no longer be held in bytes and arrays of 256 bytes cannot be used to hold a set of flags.

We have done nothing analogous to ANSI C's locales, partly because we do not feel qualified to define locales and partly because we remain unconvinced of that model for dealing with the problems. That is really more an issue of internationalization than conversion to a larger character set; on the other hand, because we have chosen a single character set that encompasses most languages, some of the need for

locales is eliminated. (We have a utility, `tcs`, that translates between UTF and other character sets.)

There are several reasons why our library does not follow the ANSI design for wide and multi-byte characters. The ANSI model was designed by a committee, untried, almost as an afterthought, whereas we wanted to design as we built. (We made several major changes to the interface as we became familiar with the problems involved.) We disagree with ANSI C's handling of invalid multi-byte sequences. Also, the ANSI C library is incomplete: although it contains some crucial routines for handling wide and multi-byte characters, there are some serious omissions. For example, our software can exploit the fact that UTF preserves ASCII characters in the byte stream. We could remove that assumption by replacing all calls to `strchr` with `utfrune` and so on. (Because of the weaker properties of the original UTF, we have actually done so.) ANSI C cannot: the standard says nothing about the representation, so portable code should *never* call `strchr`, yet there is no ANSI equivalent to `utfrune`. ANSI C simultaneously invalidates `strchr` and offers no replacement.

Finally, ANSI did nothing to integrate wide characters into the I/O system: it gives no method for printing wide characters. We therefore needed to invent some things and decided to invent everything. In the end, some of our entry points do correspond closely to ANSI routines—for example `chartorune` and `runetochar` are similar to `mbtowc` and `wctomb`—but Plan 9's library defines more functionality, enough to write real applications comfortably.

Converting the tools

The source for our tools and applications had already been converted to work with Latin-1, so it was '8-bit safe', but the conversion to Unicode and UTF is more involved. Some programs needed no change at all: `cat`, for instance, interprets its argument strings, delivered in UTF, as file names that it passes uninterpreted to the `open` system call, and then just copies bytes from its input to its output; it never makes decisions based on the values of the bytes. (Plan 9 `cat` has no options such as `-v` to complicate matters.) Most programs, however, needed modest change.

It is difficult to find automatically the places that need attention, but `grep` helps. Software that uses the libraries conscientiously can be searched for calls to library routines that examine bytes as characters: `strchr`, `strrchr`, `strstr`, etc. Replacing these by calls to `utfrune`, `utfrrune`, and `utfutf` is enough to fix many programs. Few tools actually need to operate on runes internally; more typically they need only to look for the final slash in a file name and similar trivial tasks. Of the 170 C source programs in the top levels of `/sys/src/cmd`, only 23 now contain the word `Rune`.

The programs that *do* store runes internally are mostly those whose *raison d'être* is character manipulation: `sam` (the text editor), `sed`, `sort`, `tr`, `troff`, 8½ (the window system and terminal emulator), and so on. To decide whether to compute using runes or UTF-encoded byte strings requires balancing the cost of converting the data when read and written against the cost of converting relevant text on demand. For programs such as editors that run a long time with a relatively constant dataset, runes are the better choice. There are space considerations too, but they are more complicated: plain ASCII text grows when converted to runes; UTF-encoded Japanese shrinks.

Again, it is hard to automate the conversion of a program from `chars` to `Runes`. It is not enough just to change the type of variables; the assumption that bytes and characters are equivalent can be insidious. For instance, to clear a character array by

```
memset(buf, 0, BUFSIZE)
```

becomes wrong if `buf` is changed from an array of `chars` to an array of `Runes`. Any program that indexes tables based on character values needs rethinking. Consider `tr`, which originally used multiple 256-byte arrays for the mapping. The naïve conversion would yield multiple 65536-rune arrays. Instead Plan 9 `tr` saves space by building in effect a run-encoded version of the map.

`Sort` has related problems. The cooperation of UTF and `strcmp` means that a simple sort—one with no options—can be done on the original UTF strings using `strcmp`. With sorting options enabled, however, `sort` may need to convert its input to runes: for example, option `-t`α requires searching for alphas in the input text to crack the input into fields. The field specifier `+3.2` refers to 2 runes beyond the third field. Some of the other options are hopelessly provincial: consider the case-folding and dictionary

order options (Japanese doesn't even have an official dictionary order) or `-M` which compares by case-insensitive English month name. Handling these options involves the larger issues of internationalization and is beyond the scope of this paper and our expertise. Plan 9 `sort` works sensibly with options that make sense relative to the input. The simple and most important options are, however, usually meaningful. In particular, `sort` sorts UTF into the same order that `look` expects.

Regular expression-matching algorithms need rethinking to be applied to UTF text. Deterministic automata are usually applied to bytes; converting them to operate on variable-sized byte sequences is awkward. On the other hand, converting the input stream to runes adds measurable expense and the state tables expand from size 256 to 65536; it can be expensive just to generate them. For simple string searching, the Boyer-Moore algorithm works with UTF provided the input is guaranteed to be only valid UTF strings; however, it does not work with the old UTF encoding. At a more mundane level, even character classes are harder: the usual bit-vector representation within a non-deterministic automaton is unwieldy with 65536 characters in the alphabet.

We compromised. An existing library for compiling and executing regular expressions was adapted to work on runes, with two entry points for searching in arrays of runes and arrays of chars (the pattern is always UTF text). Character classes are represented internally as runs of runes; the reserved Unicode value `FFFF` marks the end of the class. Then *all* utilities that use regular expressions—editors, `grep`, `awk`, etc.—except the shell, whose notation was grandfathered, were converted to use the library. For some programs, there was a concomitant loss of performance, but there was also a strong advantage. To our knowledge, Plan 9 is the only Unix-like system that has a single definition and implementation of regular expressions; patterns are written and interpreted identically by all the programs in the system.

A handful of programs have the notion of character built into them so strongly as to confuse the issue of what they should do with UTF input. Such programs were treated as individual special cases. For example, `wc` is, by default, unchanged in behavior and output; a new option, `-r`, counts the number of correctly encoded runes—valid UTF sequences—in its input; `-b` the number of invalid sequences.

It took us several months to convert all the software in the system to Unicode and the old UTF. When we decided to convert from that to the new UTF, only three things needed to be done. First, we rewrote the library routines to encode and decode the new UTF. This took an evening. Next, we converted all the files containing UTF to the new encoding. We wrote a trivial program to look for non-ASCII bytes in text files and used a Plan 9 program called `tcs` (translate character set) to change encodings. Finally, we recompiled all the system software; the library interface was unchanged, so recompilation was sufficient to effect the transformation. The second two steps were done concurrently and took an afternoon. We concluded that the actual encoding is relatively unimportant to the software; the adoption of large characters and a byte-stream encoding *per se* are much deeper issues.

Graphics and fonts

Plan 9 provides only minimal support for plain text terminals. It is instead designed to be used with all character input and output mediated by a window system such as 8½. The window system and related software are responsible for the display of UTF text as Unicode character images. For plain text, the window system must provide a user-settable *font* that provides a (possibly empty) picture for each Unicode character. Fancier applications that use bold and Italic characters need multiple fonts storing multiple pictures for each Unicode value. All the issues are apparent, though, in just the problem of displaying a single image for each character, that is, the Unicode equivalent of a plain text terminal. With 128 or even 256 characters, a font can be just an array of bitmaps. With 65536 characters, a more sophisticated design is necessary. To store the ideographs for just Japanese as 16×16×1 bit images, the smallest they can reasonably be, takes over a quarter of a megabyte. Make the images a little larger, store more bits per pixel, and hold a copy in every running application, and the memory cost becomes unreasonable.

The structure of the bitmap graphics services is described at length elsewhere [Pike91]. In summary, the memory holding the bitmaps is stored in the same machine that has the display, mouse, and keyboard: the terminal in Plan 9 terminology, the workstation in others'. Access to that memory and associated services is provided by device files served by system software on the terminal. One of those files, `/dev/bitblt`, interprets messages written upon it as requests for actions corresponding to entry points in the graphics library: allocate a bitmap, execute a raster operation, draw a text string, etc. The window

system acts as a multiplexer that mediates access to the services and resources of the terminal by simulating in each client window a set of files mirroring those provided by the system. That is, each window has a distinct `/dev/mouse`, `/dev/bitblt`, and so on through which applications drive graphical input and output.

One of the resources managed by 8½ and the terminal is the set of active *subfonts*. Each subfont holds the bitmaps and associated data structures for a sequential set of Unicode characters. Subfonts are stored in files and loaded into the terminal by 8½ or an application. For example, one subfont might hold the images of the first 256 characters of the Unicode space, corresponding to the Latin-1 character set; another might hold the standard phonetic character set, Unicode characters with value 0250 to 02A8. These files are collected in directories corresponding to typefaces: `/lib/font/bit/pelm` contains the Pellucida Monospace character set, with subfonts holding the Latin-1, Greek, Cyrillic and other components of the typeface. A suffix on subfont files encodes (in a subfont-specific way) the size of the images: `/lib/font/bit/pelm/latin1.9` contains the Latin-1 Pellucida Monospace characters with lower case letters 9 pixels high; `/lib/font/bit/jis/jis5400.16` contains 16-pixel high ideographs starting at Unicode value 5400.

The subfonts do not identify which portion of the Unicode space they cover. Instead, a font file, in plain text, describes how to assemble subfonts into a complete character set. The font file is presented as an argument to the window system to determine how plain text is displayed in text windows and applications. Here is the beginning of the font file `/lib/font/bit/pelm/jis.9.font`, which describes the layout of a font covering that portion of Unicode for which we have characters of typical display size, using Japanese characters to cover the Han space:

```
18      14
0x0000  0x00FF  latin1.9
0x0100  0x017E  latineur.9
0x0250  0x02E9  ipa.9
0x0386  0x03F5  greek.9
0x0400  0x0475  cyrillic.9
0x2000  0x2044  ../misc/genpunc.9
0x2070  0x208E  supsub.9
0x20A0  0x20AA  currency.9
0x2100  0x2138  ../misc/letterlike.9
0x2190  0x21EA  ../misc/arrows
0x2200  0x227F  ../misc/math1
0x2280  0x22F1  ../misc/math2
0x2300  0x232C  ../misc/tech
0x2500  0x257F  ../misc/chart
0x2600  0x266F  ../misc/ding
0x3000  0x303f  ../jis/jis3000.16
0x30a1  0x30fe  ../jis/katakana.16
0x3041  0x309e  ../jis/hiragana.16
0x4e00  0x4fff  ../jis/jis4e00.16
0x5000  0x51ff  ../jis/jis5000.16
...
```

The first two numbers set the interline spacing of the font (18 pixels) and the distance from the baseline to the top of the line (14 pixels). When characters are displayed, they are placed so as best to fit within those constraints; characters too large to fit will be truncated. The rest of the file associates subfont files with portions of Unicode space. The first four such files are in the Pellucida Monospace typeface and directory; others reside in other directories. The file names are relative to the font file's own location.

There are several advantages to this two-level structure. First, it simultaneously breaks the huge Unicode space into manageable components and provides a unifying architecture for assembling fonts from disjoint pieces. Second, the structure promotes sharing. For example, we have only one set of Japanese characters but dozens of typefaces for the Latin-1 characters, and this structure permits us to store only one copy of the Japanese set but use it with any Roman typeface. Also, customization is easy. English-speaking users who don't need Japanese characters but may want to read an on-line Oxford English

Dictionary can assemble a custom font with the Latin-1 (or even just ASCII) characters and the International Phonetic Alphabet (IPA). Moreover, to do so requires just editing a plain text file, not using a special font editing tool. Finally, the structure guides the design of caching protocols to improve performance and memory usage.

To load a complete Unicode character set into each application would consume too much memory and, particularly on slow terminal lines, would take unreasonably long. Instead, Plan 9 assembles a multi-level cache structure for each font. An application opens a font file, reads and parses it, and allocates a data structure. A message written to `/dev/bitblt` allocates an associated structure held in the terminal, in particular, a bitmap to act as a cache for recently used character images. Other messages copy these images to bitmaps such as the screen by loading characters from subfonts into the cache on demand and from there to the destination bitmap. The protocol to draw characters is in terms of cache indices, not Unicode character number or UTF sequences. These details are hidden from the application, which instead sees only a subroutine to draw a string in a bitmap from a given font, functions to discover character size information, and routines to allocate and to free fonts.

As needed, whole subfonts are opened by the graphics library, read, and then downloaded to the terminal. They are held open by the library in an LRU-replacement list. Even when the program closes a subfont, it is retained in the terminal for later use. When the application opens the subfont, it asks the terminal if it already has a copy to avoid reading it from the file server if possible. This level of cache has the property that the bitmaps for, say, all the Japanese characters are stored only once, in the terminal; the applications read only size and width information from the terminal and share the images.

The sizes of the character and subfont caches held by the application are adaptive. A simple algorithm monitors the cache miss rate to enlarge and shrink the caches as required. The size of the character cache is limited to 2048 images maximum, which in practice seems enough even for Japanese text. For plain ASCII-like text it naturally stays around 128 images.

This mechanism sounds complicated but is implemented by only about 500 lines in the library and considerably less in each of the terminal's graphics driver and 8½. It has the advantage that only characters that are being used are loaded into memory. It is also efficient: if the characters being drawn are in the cache the extra overhead is negligible. It works particularly well for alphabetic character sets, but also adapts on demand for ideographic sets. When a user first looks at Japanese text, it takes a few seconds to read all the font data, but thereafter the text is drawn almost as fast as regular text (the images are larger, so draw a little slower). Also, because the bitmaps are remembered by the terminal, if a second application then looks at Japanese text it starts faster than the first.

We considered building a 'font server' to cache character images and associated data for the applications, the window system, and the terminal. We rejected this design because, although isolating many of the problems of font management into a separate program, it didn't simplify the applications. Moreover, in a distributed system such as Plan 9 it is easy to have too many special purpose servers. Making the management of the fonts the concern of only the essential components simplifies the system and makes bootstrapping less intricate.

Input

A completely different problem is how to type Unicode characters as input to the system. We selected an unused key on our ASCII keyboards to serve as a prefix for multi-keystroke sequences that generate Unicode characters. For example, the character ü is generated by the prefix key (typically `ALT` or `Compose`) followed by a double quote and a lower-case u. When that character is read by the application, from the file `/dev/cons`, it is of course presented as its UTF encoding. Such sequences generate characters from an arbitrary set that includes all of Latin-1 plus a selection of mathematical and technical characters. An arbitrary Unicode character may be generated by typing the prefix, an upper case X, and four hexadecimal digits that identify the Unicode value.

These simple mechanisms are adequate for most of our day-to-day needs: it's easy to remember to type 'ALT 1 2' for ½ or 'ALT accent letter' for accented Latin letters. For the occasional unusual character, the cut and paste features of 8½ serve well. A program called (perhaps misleadingly) `unicode` takes as argument a hexadecimal value, and prints the UTF representation of that character, which may then be

picked up with the mouse and used as input.

These methods are clearly unsatisfactory when working in a non-English language. In the native country of such a language the appropriate keyboard is likely to be at hand. But it's also reasonable—especially now the system handles Unicode—to work in a language foreign to the keyboard.

For alphabetic languages such as Greek or Russian, it is straightforward to construct a program that does phonetic substitution, so that, for example, typing a Latin 'a' yields the Greek 'α'. Within Plan 9, such a program can be inserted transparently between the real keyboard and a program such as the window system, providing a manageable input device for such languages.

For ideographic languages such as Chinese or Japanese the problem is harder. Native users of such languages have adopted methods for dealing with Latin keyboards that involve a hybrid technique based on phonetics to generate a list of possible symbols followed by menu selection to choose the desired one. Such methods can be effective, but their design must be rooted in information about the language unknown to non-native speakers. (`Cxterm`, a Chinese terminal emulator built by and for Chinese programmers, employs such a technique [Pong and Zhang].) Although the technical problem of implementing such a device is easy in Plan 9—it is just an elaboration of the technique for alphabetic languages—our lack of familiarity with such languages has restrained our enthusiasm for building one.

The input problem is technically the least interesting but perhaps emotionally the most important of the problems of converting a system to an international character set. Beyond that remain the deeper problems of internationalization such as multi-lingual error messages and command names, problems we are not qualified to solve. With the ability to treat text of most languages on an equal footing, though, we can begin down that path. Perhaps people in non-English speaking countries will consider adopting Plan 9, solving the input problem locally—perhaps just by plugging in their local terminals—and begin to use a system with at least the capacity to be international.

Acknowledgements

Dennis Ritchie provided consultation and encouragement. Bob Flandrena converted most of the standard tools to UTF. Brian Kernighan suffered cheerfully with several inadequate implementations and converted `troff` to UTF. Rich Drechsler converted his Postscript driver to UTF. John Hobby built the Postscript ☺. We thank them all.

References

[ANSIC] *American National Standard for Information Systems – Programming Language C*, American National Standards Institute, Inc., New York, 1990

[ISO10646] ISO/IEC DIS 10646-1:1993 *Information technology – Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, ''Plan 9 from Bell Labs'', UKUUG Proc. of the Summer 1990 Conf., London, England, 1990

[Pike91] Pike, R., ''8.5, The Plan 9 Window System'', USENIX Summer Conf. Proc., Nashville, 1991

[Pike92] Pike, R., ''How to Use the Plan 9 C Compiler'', in *The Plan 9 Programmer's Manual*, AT&T Bell Laboratories, Murray Hill, NJ, 1992

[Pong and Zhang] Man-Chi Pong and Yongguang Zhang, ''cxterm: A Chinese Terminal Emulator for the X Window System'', *Software–Practice and Experience,* Vol 22(1), 809-926, October 1992.

[Unicode] *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, The Unicode Consortium, Addison Wesley, New York, 1991

NAME

UTF, Unicode, ASCII, rune – character set and format

DESCRIPTION

The Plan 9 character set and representation are based on Unicode and on a proposed X-Open multibyte FSS-UCS-TF (File System Safe Universal Character Set Transformation Format) encoding. Unicode represents its characters in 16 bits; FSS-UCS-TF, or just UTF, represent such values in an 8-bit byte stream.

In Plan 9, a *rune* is a 16-bit quantity representing a Unicode character. Internally, programs may store characters as runes. However, any external manifestation of textual information, in files or at the interface between programs, uses a machine-independent, byte-stream encoding called UTF.

UTF is designed so the 7-bit ASCII set (values hexadecimal 00 to 7F), appear only as themselves in the encoding. Runes with values above 7F appear as sequences of two or more bytes with values only from 80 to FF.

The UTF encoding of Unicode is backward compatible with ASCII: programs presented only with ASCII work on Plan 9 even if not written to deal with UTF, as do programs that deal with uninterpreted byte streams. However, programs that perform semantic processing on ASCII graphic characters must convert from UTF to runes in order to work properly with non-ASCII input. See *rune*(2).

Letting numbers be binary, a rune x is converted to a multibyte UTF sequence as follows:

01. x in [00000000.0bbbbbbb] → 0bbbbbbb
10. x in [00000bbb.bbbbbbbb] → 110bbbbb, 10bbbbbb
11. x in [bbbbbbbb.bbbbbbbb] → 1110bbbb, 10bbbbbb, 10bbbbbb

Conversion 01 provides a one-byte sequence that spans the ASCII character set in a compatible way. Conversions 10 and 11 represent higher-valued characters as sequences of two or three bytes with the high bit set. Plan 9 does not support the 4, 5, and 6 byte sequences proposed by X-Open. When there are multiple ways to encode a value, for example rune 0, the shortest encoding is used.

In the inverse mapping, any sequence except those described above is incorrect and is converted to rune 0080.

FILES

`/lib/unicode` table of characters and descriptions, suitable for *look*(1).

SEE ALSO

ascii(1), *tcs*(1), *rune*(2), *keyboard*(6), *The Unicode Standard*.