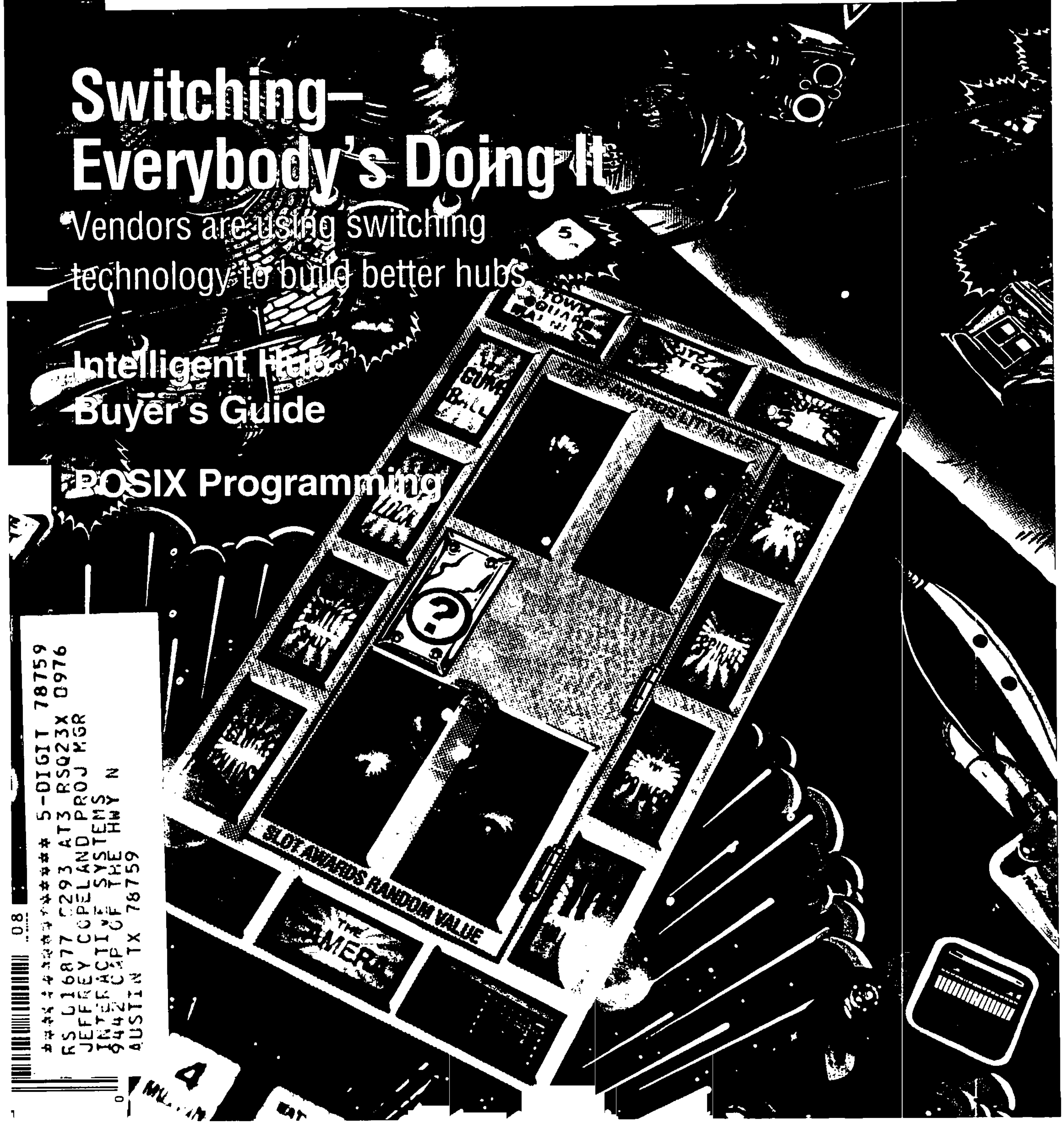
RS/Magazine

The Journal For IBM Workstation Users

VOLUME 2 NUMBER 8 $5.50 AUGUST 1993



Switching—Everybody's Doing It

Vendors are using switching technology to build better hubs

Intelligent Hub Buyer's Guide

POSIX Programming

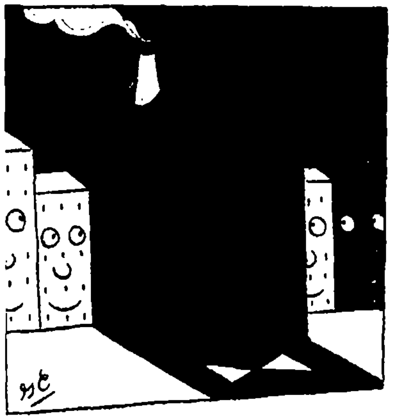
```
****************** 5-DIGIT 78759
RS L16877 C293 AT3 RSQ23X 0976
JEFFREY COPELAND PROJ MGR
INTERACTIVE SYSTEMS
9442 CAP OF THE HWY N
AUSTIN TX 78759
```

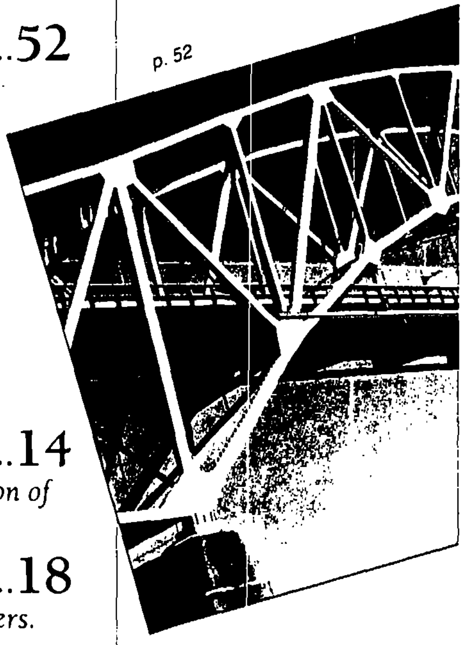
AUGUST 1993

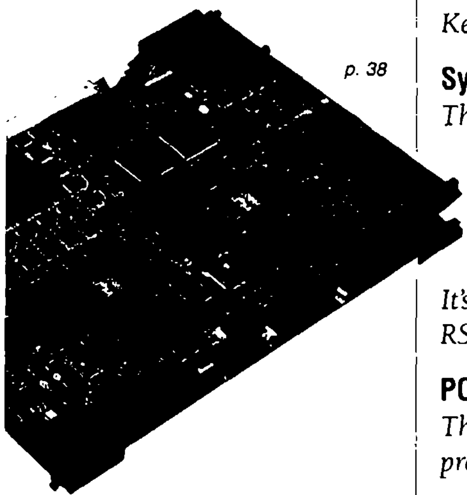
VOLUME 2 NUMBER 8

RS/Magazine

The Journal For IBM Workstation Users


p. 28

p. 52


p. 38


columns illustrated by GREG CLARKE
cover photograph ©1993 TOM CROKE/VISUAL IMAGE INC.
pinball machine courtesy of NEW ENGLAND COIN-OP DISTRIBUTING INC.

RS/Magazine is not affiliated with IBM Corp.

RS/Magazine (ISSN 1061-0030) is published monthly by Computer Publishing Group Inc., 1330 Beacon St. Brookline, MA 02146 Telephone (617) 739-7001 Second-class Postage Rates paid at Boston, MA, and at additional mailing offices. Posted under Canadian IPM #0235881. This publication is free to qualified subscribers as determined by the publisher. Subscription rates are $60 per year in the United States, and $95 (surface mail) and $150 (airmail) outside the United States. Subscription requests can be sent to Circulation Department, RS/Magazine, 1330 Beacon St, Brookline, MA 02146 or electronically mailed to circ@expert.com. POSTMASTER, please send all address changes to RS/Magazine, Circulation Department, 1330 Beacon St., Brookline, MA 02146. Please allow 6-8 weeks for change of address. Include your old address as well as new–enclosing, if possible, an address label from a recent issue. All rights reserved © Copyright 1993, Computer Publishing Group Inc. No part of this publication may be transmitted or reproduced in any form by any means without permission in writing from the publisher. Material for publication should be sent to the attention of Anne Knowles at the above address or electronically mailed to aknowles@expert.com. Letters sent to the publication become the property of the publication and are assumed to be intended for publication and may be used so. All information herein is believed to be accurate to the best of our ability.

RS/Magazine
The Journal For IBM Workstation Users

publisher
S. HENRY SACKS
editorial director
DOUGLAS PRYOR
editor
ANNE KNOWLES
managing editor
LISA GUISBOND
art director
JOHN W. KELLEY JR.
senior editor
JANE MAJKIEWICZ
technical editor
BARRY SHEIN
research editor
MAUREEN MCKEON
assistant managing editor
MARY ANNE WEEKS MAYO
marketing manager
SUSAN R. SACKS
associate art director
HANNA DYER
designer
LEE A. BARTELL
production director
RICHARD M. ABAID
production manager
DEBORAH BEDLOW
circulation director
DEBORAH MOORE
circulation assistant
PATRICK T. COLEMAN
administrative assistant
TINA JACKSON

EDITORIAL OFFICES
1330 BEACON STREET
BROOKLINE, MA 02146
(617) 739-7001
email: aknowles@expert.com

BPA International Business Publication Membership Applied for December 1992
Printed in USA

Editorial

IBM's Commercial Legacy

IBM's historic influence in the commercial market bodes well for the future of its RS/6000 line. It couldn't bode badly, though IBM's clout isn't quite what it used to be. According to IBM, sales of RS/6000s destined for commercial applications are growing 30% to 35% annually and will soon surpass technical sales, if that hasn't happened already. The future of IBM's competitors–i.e., Sun and HP–lies in the $10 billion-plus commercial UNIX market as well, especially in their collective quest for a piece of the bigger PC pie. (For more on IBM's plans to support AIX and PC operating systems on a single platform, see RS/News, "IBM's All-purpose OS").

IBM's commercial division recently did a dog-and-pony show to outline its plans and product line to the press. The group's disclosures included plans to bring out multiple versions of AIX for different platforms and applications (see RS/News, "AIX Flavors"), plans for its hardware line that will span notebooks to message-passing parallel processors with hundreds of processors (see AIXtensions for discussion on some of the larger systems) and the announcement of a smaller but dedicated sales force selling the RS/6000.

IBM, for example, is in the process of commercializing the SP1 parallel system with beefed-up I/O, the CICS transaction-processing monitor and a version of AIX that has MVS attributes. Big Blue certainly understands the data-center environment and data-server concept better than its rivals. Yet the company still seems to suffer from another legacy that does not bode well for its future–excessive layers of decision making.

Also, in enterprisewide networks, hubs are playing an increasingly important role, especially as their intelligence and functionality grow. This month's cover story, written by Senior Editor Jane Majkiewicz, explains the different switching technologies used by most hub vendors. The buyer's guide, compiled by Research Editor Maureen McKeon, lists intelligent hubs offered by 28 manufacturers. (If IBM didn't make the list, it's not our fault. Their many decision makers didn't meet the deadline.) This month, we also introduce a new column by our I18N duo. This time around, Jeffreys Copeland & Haemer are tackling POSIX programming, from the outside in.

[illegible]

From the Outside In

by Jeffreys Copeland and Haemer

This month, we start a series on POSIX programming. This first column is called "From the Outside In" because we plan to approach POSIX.1 from an unorthodox direction: We'll start with shell-level commands, which we assume you know, and try to infer what must be in POSIX.1–the programming interfaces to the operating system–to make the commands work. We assume you have some knowledge of Standard C and some familiarity with a UNIX system. If you're reading this magazine, that's probably AIX, which is fine since AIX on the RS/6000 was designed to be POSIX-conforming.

This column will provide an overview: In a sort of question-and-answer forum, we'll discuss what's in POSIX.1 and how it fits into the world of standards. In later columns, we'll tackle the details. Along the way, we'll digress a lot to talk about portability, style, design and whatever else takes our fancy but always come back to POSIX.1, the lynchpin of UNIX systems programming. The columns are based on one of Canary Software's courses on POSIX programming.

Why POSIX?

Over the past few years, vendors and users have been moving toward standards-based computing. The phrase most heard has been "open systems," which, after all the marketing hype, has become as empty as the word "best-seller" in the publishing business or "blockbuster" in the movie industry. The impetus behind open systems is that, though hardware is cheap and getting cheaper, software–building, maintaining and learning to use software–is expensive and getting even more expensive. We can't afford to throw it out, so we have to pay to port it.

The big breakthrough in porting was UNIX, the first real, portable operating system. Invented by Ken Thompson in 1967, UNIX was running on CPUs as disparate as the 8088 and the Cray 1 by the mid-

Jeffrey Copeland (jeff@aus.shl.com) *lives in Austin, TX, where he is project manager for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

1980s. Dennis Ritchie's C, written so that he could create UNIX, had spread even farther.

Porting system software from one UNIX to another was easy–well, not exactly easy, but not appreciably more difficult than porting a FORTRAN application from one vendor's compiler to another. As Steve Johnson pointed out about the portable C compiler back when UNIX Version 6 was the only game in town: Each time you port a piece of software, it takes roughly half the effort of the last port.

The key UNIX question became how to cut porting costs. A piece of the answer to this problem in the software industry has always been "standards": Figure out what everyone agrees on, standardize the syntax and semantics of that core and move on. Language standards, such as FORTRAN-66, FORTRAN-77, FORTRAN-90 and an endless succession of COBOL standards, provide familiar examples of this. But a standard for an operating system?

Almost any OS can conform to POSIX standards.

The first group to seize this new idea and actually do something with it was UniForum (then /usr/group), which created a "standard" based on AT&T's UNIX System III. And just as programming-language standards avoid specifying compiler-implementation details, the /usr/group standard didn't try to standardize UNIX internals, it specified interfaces: the section 2 and 3 calls of traditional UNIX manuals.

Although it had no official standing–/usr/group not being an accredited standards-making body–the /usr/group standard was quickly adopted by the IEEE as a base document for a real, operating-system-standards project. Moved along by pressure from the National Bureau of Standards, powered by the U. S. government's enormous computer purchases, the POSIX standards project quickly grew to enjoy nearly universal industry support. With almost unheard-of speed, POSIX produced ANSI/IEEE Standard 1003.1 in 1988 and its revision, ISO Standard IS 9945-1, in 1990.

That's just the beginning. There are now nearly two dozen POSIX standards projects that range from standards for other pieces of the operating system, such as the shell and shell-level commands, to system-level APIs for other languages, such as Ada. In our articles, we'll confine ourselves to the original POSIX.1 standard because we don't want to bite off more than we can chew. Except, of course, when we feel like it.

So, is POSIX UNIX?

No, it's not. POSIX is an acronym for a family of standards whose full title is Portable Operating System Interface. (Actually, it isn't an acronym, since IX doesn't stand for interface. POSIX is a name invented by Richard Stallman of the Free Software Foundation.) In principle, almost any operating system can conform to POSIX standards. For example, the current off-the-shelf release of DEC's VMS provides the POSIX.1 system call interfaces.

We say "almost" because it isn't always possible to conform. You just can't `fork()` on MS-DOS. This is our lead-in to the point that the various standards in the POSIX suite are modeled on UNIX but they need not come bundled as a package. Mortice-Kern Systems provides a nearly complete set of POSIX.2 utilities (the shell and shell-level commands) for MS-DOS. Installing these on your DOS laptop won't help you port your C code to it, but a carefully written shell script should work just as well on your 286 above the clouds in coach class as it does on your RS/6000 at home.

How did POSIX come into being?

This is almost the "why is the sky blue?" question for the month, and the answer is almost as long-winded. (Side note to our kids: The sky is blue because of scattering, which causes the air to act like a prism. For more details, read section 32-5 of *The Feynman Lectures on Physics*, Volume I, before you go to bed tonight.)

One good approach to the answer is to explore the twisty maze of little standards bodies. We've already mentioned UniForum, an organization that traditionally helps identify areas ripe for standardization. Such recommendations are made to approved standards-making organizations, such as ANSI's X3 committee and the IEEE. These organizations don't actually make standards themselves, but delegate the job to subcommittees populated by volunteer technical experts. In this case, the IEEE gave the job to the Portable Applications Standards Committee (PASC), and its subcommittees, 1003.1, 1003.2, 1003.3, 1003.3.1, 1003.3.2... The numbers are arbitrary and roughly chronological. So far as we know, the numbers have to be rational, even if the committees don't; otherwise, the supercomputing group could have been 1003.π. They probably even have to be nonnegative; there is a 1003.0, perhaps in testament to UNIX's ties to C, but no 1003.-1.

In the next tier of the bleachers, we have the American National Standards Institute (ANSI) and the International Standards Organization (ISO), which approve standards and make them... well, standards. In the box seats are groups such as χ/Open and the

National Institute of Standards and Technology (NIST), which select standards to adopt and promote. Finally, out on the proverbial level playing field, we have vendors and consortia, such as UNIX International and the Open Software Foundation (OSF), which take the forests worth of confetti, er, paper that comes out of the stands and standards, and turn them into real products.

Alternately, since the knee bone's connected to the–unh!–thigh bone, we can retraverse some of these bodies from the bottom up. POSIX.1 was created by volunteers acting under the authority of IEEE's PASC. PASC is one of the standards committees of the IEEE Computer Society that creates electrical and electronics standards for the United States covering everything from Ethernets to three-phase power. The IEEE gets its authority from ANSI, which standardizes everything from screw threads to motorcycle helmets. (Take a look at the little sticker on the next pair of sunglasses you buy to see what ANSI standard they conform to.) ANSI approves U.S. standards under the authority delegated to them by ISO to make U.S. national standards. ISO, originally chartered by the United Nations, can promote national standards submitted by its member bodies, such as ANSI, British Standards Institute (BSI), Association Française de Normalisation (AFNOR) and others, to international standards. The DIN number on your film boxes is from the German standards institute–Deutsches Institut für Normalization–which also has standards for trash cans. We wonder if the Mac icon conforms.

Out of this multilevel hierarchy of organizations involved in getting a standard created, POSIX has ended up being the central standard for operating systems.

Out of this multilevel heirarchy, POSIX has ended up being the central standard for operating systems.

Well, what's in POSIX.1?

Good question. Mostly, POSIX.1 contains function-call interfaces Think of it as the man pages for traditional system interfaces, such as `read()` and `chdir()`, made uniform and unambiguous, and translated into standards-ese. There's more in the standard–header files, data types, symbolic constants and such–but most of these are things that the standard needs to define in order to let implementors and application programmers–the customers of the standard–provide and use the interfaces. For example, it doesn't make sense to define `open()` or `read()` without also defining the data types they use for arguments, the data type they return and the headers that contain these declarations and the function prototypes.

Another good question is, "Well, what's *not* in POSIX.1?" We've mentioned some things already, but here's a more complete list.

Other language bindings aren't. What are language bindings? That's what we use to call particular functions from particular languages. POSIX.1 currently only provides C calls for functions. POSIX.5 and POSIX.9 are the corresponding functions for Ada and FORTRAN. For example, POSIX.1's `pipe()` is POSIX.9's `SUBROUTINE PXFPIPE()`.

Things that aren't interfaces aren't. The POSIX.1 standard says nothing about hardware, such as byte order or keyboard labels, nor about internal implementation details like the internals of a directory or how the file system is laid out. Such omissions let us implement POSIX on top of completely alien operating systems, such as CTOS or Windows NT.

Commands aren't. They're in POSIX.2. In fact, a whole series of interfaces was omitted from POSIX.1 either because there was too little consensus (system administration), or because the interfaces were thought to be a sideshow that should be delegated to more specialized groups (real-time, supercomputing).

Files aren't. Believe it or not, there is no `/tmp`, no `/etc/passwd`, no `/dev/null`. Sometimes there are access functions, such as `getpwent()`, that return the information in such files, but individual filenames are absent from the standard. Remember, POSIX isn't UNIX, and non-UNIX implementations may have different names for these ideas. Contrast this with, for example, the SVID, which specifies that most temporary files should be placed in `/usr/tmp`. Our Berkeley Software Design Inc. (BSDI) machine doesn't even have a `/usr/tmp`.

Users aren't. Not even the ubiquitous UNIX user, `root`. POSIX refers instead to users with "appropriate privilege." We used to think this meant users who had gone to a New England prep school, but it actually means the superuser without coming right out and saying it. Again, POSIX isn't UNIX. On VMS this user is called `SYSTEM` instead of `root`. This circumlocution also lets vendors produce an Orange-Book secure version of POSIX, with UNIX's traditional root privilege split up among different IDs so that permission to back up the password file need not imply permission to peruse it.

Version-specific features aren't. Sometimes this is okay, as anyone who has complained about or praised AIX-specific features can agree. Sometimes it's an annoying and out-of-date compromise. Symbolic links, for example, are still absent from POSIX.1.

What are these interfaces you keep talking about?

Traditional UNIX separates the idea of system calls–such as `read()` or `seek()`, which you have to ask the operating system to do–from library calls–such as `getchar()` and `fseek()`, which you can implement in user-level code using the more primitive system calls. POSIX doesn't bother with this distinction, for two reasons. On one hand, we programmers usually don't care whether it's implemented in the kernel or not. On the other, one system's library routine may be another's system call. Moreover what's a system call can change over time. Routines like `creat()`, `dup()` and `dup2()` started out as system calls. Today, `creat()` is more likely to be a library function that calls the new, three-argument `open()`, and `dup()` and `dup2()` are probably just user-level routines that call `fcntl()`.

Rather than stumbling over the cumbersome phrase "library routine or system call," POSIX just calls them all "interfaces."

How does Standard C come into this?

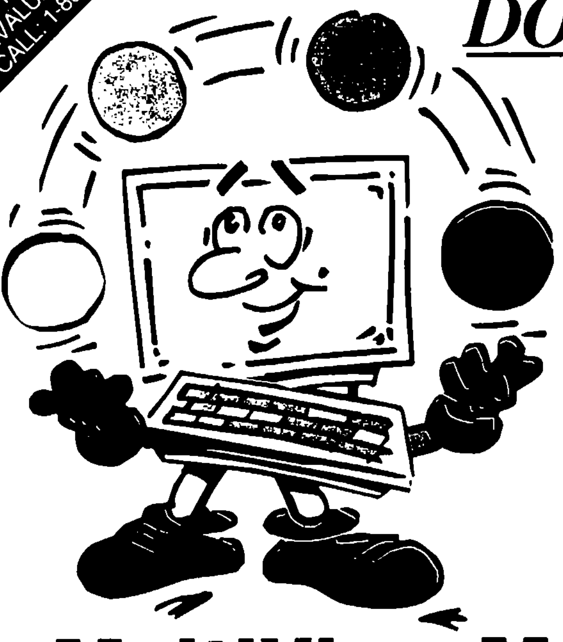
How indeed? Well, at first it couldn't, for the most pedestrian of reasons: Even though the POSIX.1 work started after the Standard C work, it finished first. Can't very well refer to a standard that doesn't exist yet, can you?

By the time the second, and international, POSIX incarnation rolled around in 1990, there was a perfectly good C standard, but not everyone had switched over to it, so the 1990 standard still couldn't really afford to tie itself to Standard C. Still, it was clear to most folks that Standard C was the way to go where possible, so POSIX.1 made an odd, but sensible, compromise: Implementations and applications could be in either Standard C or "Common-Usage C," but common-usage implementations and applications were constrained in ways that would ease a future migration. For example, even common-usage C implementations had to supply a number of functions, data types, header files and limits defined by Standard C.

But there's more interconnections. Because Standard C was designed to be implemented on a wide variety of non-UNIX platforms, like MS-DOS

SEND TODAY FOR YOUR **FREE** EVALUATION COPY CALL: 1-800-359-3408

Let your terminal
<u>DO MORE</u>
than one thing at a time with...



MultiView Mascot

Does using a terminal slow your work down?

Make your terminal behave like a PC by using ***MultiView Mascot*** to deliver the benefits of ***windowing***.

Adding ***MultiView Mascot*** lets you accomplish more with your existing applications, simply by letting your terminal do more than one thing at a time.

With ***MultiView Mascot*** you can:

- ***Run multiple applications at once***
- ***Switch rapidly between them***
- ***Cut & Paste information from one application to another***

Without changing your application, add the benefits of windowing to increase productivity and reduce frustration.

To find out how you can do more with MultiView Mascot simply call JSB today and watch your productivity soar.

NOW SUPPORTS
***ALPHAWINDOW* TERMINALS**

Call: 1-800-359-3408

JSB

The Americas JSB CORPORATION, 108 Whispering Pines Drive, Suite 115, Scotts Valley, California 95066, USA. Telephone (408) 438 3800 Facsimile: (408) 438 8360

Europe: JSB CORPORATION, Cheshire House, Castle Street, Macclesfield, Cheshire. SK11 6AF. England Telephone: +44 (0) 625 433618 Facsimile +44 (0) 625 433948

All Trademarks Recognized

Circle No. 15 on Inquiry Card

Table

1003.0	POSIX guide
1003.1	System services interface
1003.2	Shell and tools
1003.3	Test methods
1003.4	Real-time (includes threads)
1003.5	Ada bindings
1003.6	Security
1003.7.1	Print administration
1003.7.2	Software administration
1003.8	Transparent file access
1003.9	FORTRAN bindings
1003.10	Supercomputing AEP
1003.11	Transaction processing
1003.12	Protocol-independent interface
1003.13	Real-time AEP
1003.14	Multiprocessing
1003.15	Supercomputing batch
1003.16	C binding for .1 LIS
1003.17	Name space/directory services (X.500)
1003.18	POSIX environment profile
1003.19	FORTRAN-90 binding to .1 1003.20, Ada binding to .4
1201.1	Windowing tool kit
1201.2	Drivability
1224	Object management (ASN.1)
1224.1	X.400 API
1238.0	OSI API connection management
1238.1	FTAM API

and VMS, many traditional UNIX interfaces, such as `open()`, `link()` and `read()`, were omitted and some new interfaces, such as `rename()`, weren't specific about their behavior on operating-system-specific structures, such as directories. POSIX.1 had to plug these holes, but in ways that didn't break Standard C. Reading the standard with this problem in mind sometimes helps it make more sense. Meanwhile, events undercut some of the foundations of this reasoning.

First, so many vendors clambered on the POSIX bandwagon so quickly that the Ada and FORTRAN communities saw there would be real benefit to creating their own bindings to POSIX systems. If all operating systems were going to supply functions such as `link()` and `getpwd()`, why shouldn't FORTRAN programmers have a standard interface to them, too? These standards were created quickly and ably by PASC subcommittees, and are available as ANSI/IEEE Standards 1003.5 (Ada), and 1003.9 (FORTRAN). Other language bindings are in the works, and there has even been a controversial effort over the past few years to create a programming-language-independent version of POSIX.1 that arbitrary programming-language communities could use to turn out thin, language-specific bindings with little effort.

(The controversy has revolved around whether such a language-independent specification (LIS) is worth the effort–whether the resources used in producing an LIS are offset by the gains it might produce. If you think of standards as a tax on the industry, the question is whether to raise taxes. American engineers tend to think they shouldn't be raised without clear-cut and widespread advantages to broad sectors of the industry. Standards bureaucrats and European academics tend to think that they should be raised to "level the playing field," and remove any unfair advantage of C over, say, Mesa. The Jeffs are, we admit, American engineers.)

Second, the rapid acceptance of Standard C made it increasingly anachronistic to provide for a common-usage binding. The U. S. Government's POSIX Federal Information Procurement Standard (FIPS) 151-2, already requires the Standard C option, and the next version of POSIX.1 will probably lack the common-usage option.

We speculate that POSIX.1 may have even accelerated the dispersal and acceptance of Standard C. Enough of Standard C had to be available on platforms that only provided common-usage capabilities to lower the activation energy and provide a full-blown implementation.

If POSIX.1 is the central standard, what are the outlying ones?

The Table tothe left contains a sample snapshot of PASC activities. We've been making tables like this for years and they typically go out-of-date between the time we make them and the time we use them. We give up. This table isn't meant to be correct in detail, but it should give you a feeling for the kinds of things PASC is up to. As a rule, 1003.*n* groups are meant to specify an operating system, while other-numbered groups are meant to specify interfaces that may be found in a POSIX environment, but may be useful elsewhere as well. The distinction is arguably iffy.

Naturally, there are also relevant "standards" outside of PASC. Some, like IEEE 802.3, are important, but completely independent of POSIX. Some, however, have a closer relationship to our queen of software standards. First among these are standards pointed at by POSIX. ISO 9899, the C standard, is an obvious example. You can find a list of such standards in the POSIX.1 standard itself.

Incidentally, if you don't have a copy of the POSIX standard handy, order one. It's a working programmer's reference tool and you need one around. Order it from

IEEE Standards Office
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331
(603) 881-0480

Equally tightly tied-in are specifications that point at POSIX. Chief among these is the government's FIPS 151-2. This says what the U.S. government can and can't buy. Basically, if you want to sell a multiuser machine to Uncle Sam, your operating system has to provide POSIX.1. The last time the government was this serious about software standards was when it said that they would buy no more machines that you had to program in assembly language or machine code; if you wanted to sell to them, you had to supply a compiler for the then-new, high-level language, COBOL.

χ/Open's XPG and USL's SVID are also important specifications that require POSIX.1.

What's conformant? What's compliant?

One more pair of buzzwords. Often, people use "conformant" and "compliant" interchangeably. Is this right?

Let's turn to Webster's. First, "compliant":

com pli' ant *adj.* complying or tending to comply; yielding; submissive. –SYN. see **obedient**.

Next "conformant": Oops. There isn't any such word. Oh good. It always sounded like jargon to us. How about plain, old "conform" and its present participle "conforming"? Well, synonyms for "conform" are "adapt" and "agree." On the surface, "comply" has a far more mandatory sense than "conform": one conforms to convention if one wishes, but complies with dictates from the authorities–or else.

The historical uses of the two words in the world of standards reflects this difference. People speak of conforming to standards, a voluntary act of good software citizenship, but complying with the FIPS, for which there is a test suite and a penalty for non-compliance.

That said, there is no current, official distinction between these words. Still, we find the traditional usages helpful, and we'll try to stick to them in our columns.

What now?

Well, we're out of space and time. (There's a general relativity joke in there, but we'll skip it.) Next month we'll be back to talk about rules for headers and identifiers ▲

NOW FOR RS/6000

AFFORDABLE, FULL-FEATURED SUPPORT FOR YOUR REMOTE TERMINALS AND PRINTERS

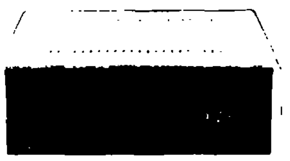
From the experts
in network management:

DataComm
for Business, Inc.

Modems, multiplexers,
DSU/CSU's everything for your
data communications application.

Call for free multiplexer information.

DataComm
for Business, Inc.



DataComm for Business, Inc.
807 Pioneer
Champaign, IL 61820

Toll-free: (800) 637-1127

In Illinois: (217) 352-3207 • Fax (217) 352-0350

Circle No. 8 on Inquiry Card

BUY • SELL • LEASE

NEW AND RECONDITIONED IBM

RS/6000
RT/6150
SERIES/1
SYSTEM 36
AS/400

IBM
Authorized
Distributor Product
Integrator

CALL:
(800) 888-2000

Dempsey
BUSINESS SYSTEMS
Where IBM Quality Is Second Nature.

18377 Beach Blvd., Suite 323
Huntington Beach, California 92646
(714) 847-8486 • FAX (714) 847-3149

IBM is a registered trademark of International Business Machines Corporation.

Circle No. 10 on Inquiry Card

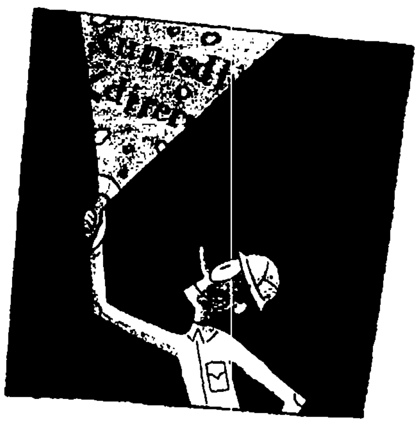
Headers, Identifiers and Writing Programs

by Jeffreys Copeland and Haemer

As discussed last time, we're planning to explore the POSIX standards from a programmer's point of view. This means we'll begin with the way commands work and deduce what the underlying features of the system must be like. Last month, we explored some of the details of how POSIX came to be, such as the ins and outs of standards committees. This month let's start with a little structure in the system, rather than outside it, and proceed to some programs.

Header Files and Identifiers

Over the years, we've come to take some things in the UNIX system for granted. File names are always a maximum of 14 characters long...or are they? And directory entries are always in the same format...except when they aren't or when file names are more than 14 bytes. Of course, we can really count on user IDs always being 16 bits long...except that we can't. So a lot of things are now defined in POSIX that weren't defined in the past and led to a lot of "diversity among" in different breeds of UNIX. Sometimes the way POSIX works around this is the way you would: A constant is defined for the appropriate limits, which are then painlessly changed when the program is recompiled on a new system.



We begin by defining some additional header files, such as `<dirent.h>`, which contains the structures for a directory entry, and the declarations of the routines to read and write them. We also define `<unistd.h>` containing the function prototypes for all the UNIX...er, POSIX functions. While we're at it, we've listed all the new include files (see Table 1).

These new header files, and some additions to the old ones, give us a

Jeffrey Copeland (`jeff@aus.shl.com`) *lives in Austin, TX, where he is project manager for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

variety of useful new data types. Just as a `sizeof(foo)` in ANSI C now returns a `size_t` instead of an `int`, a process id is no longer restricted to 16 bits, since it is now contained in a `pid_t` type. Similarly, `size_t` as defined in ANSI C does well for specifying positions within a file, but `read()` needs to return a `-1` on the end of a file. Therefore, POSIX defines a type `ssize_t`, which `read()` returns, and which can take all the same values as `size_t` as well as `-1`. (This is an example of a standards hack.)

We also have a complete set of new macro and symbolic constant names that make our life a lot easier. Instead of writing

```
lseek( fd, 0L, 0 );
```

to go to the beginning of the file, we write

```
lseek( fd, (off_t) 0, SEEK_SET );
```

where `off_t` is a new type that holds a file offset, or file size (and would be returned by, say, `lseek`), and SEEK_SET is a symbolic constant that tells `lseek()` to begin at the beginning of the file.

Earlier, we talked about the number of bytes in a file name. How do you tell what the maximum file-name length is on your system? There are a number of constants defining system limits. _NAME_MAX will tell you how big your file name can be, so that if you define an array `char fname[NAME_MAX]`, you'll be assured of having

Table 1. POSIX Header Files

<dirent.h>	structures and definitions for directory entries
<fcntl.h>	file control information
<grp.h>	group structures
<pwd.h>	password file structures
<sys/stat.h>	stat block for files
<sys/times.h>	tms structure returned by times()
<sys/types.h>	POSIX special type definitions
<sys/utsname.h>	uname structures
<sys/wait.h>	information for wait() interface
<tar.h>	definitions for tar archives
<termios.h>	terminal I/O definitions
<unistd.h>	standard POSIX definitions and prototypes
<utime.h>	access and modification times for files from utime()

Table 2. Some POSIX Constants

Name	AIX Limit	
CHAR_BIT	8	bits in a char
INT_MAX	2147483647	largest integer
CHAR_MAX	255	largest value of a byte
CHAR_MIN	0	smallest value of a byte
LONG_MAX	2147483647	largest long integer
NAME_MAX	*	file name size
_POSIX_ARG_MAX	4096	# of bytes in command line arguments
_POSIX_CHILD_MAX	6	# of child processes
_POSIX_LINK_MAX	8	# of file links
_POSIX_MAX_CANON	255	bytes in a terminal input line
_POSIX_MAX_INPUT	255	bytes in terminal input queue
_POSIX_NAME_MAX	14	bytes per file name
_POSIX_OPEN_MAX	16	open files per process
_POSIX_PATH_MAX	255	bytes per path name
_POSIX_PIPE_BUF	512	size of a pipe buffer
_POSIX_SSIZE_MAX	32767	file size
_POSIX_TZNAME_MAX	3	bytes in a time zone name

* For AIX, _NAME_MAX is only available from pathconf().

enough room. Some of the POSIX limits are in Table 2, along with their values on AIX. In general, a limit FOO is the system maximum, but the limit _POSIX_FOO is the POSIX-prescribed minimum maximum, or the minimum value the system limit can have. In other words, the smallest value that NAME_MAX can take is _POSIX_NAME_MAX or 14, so you will never find a system with a maximum file name size of less than 14 bytes. For portability, you want to set your file name buffers to at least 14, but you'll be better off using sysconf() or pathconf() to get system limits dynamically. We'll discuss these two interfaces later.

What about names? Can these new POSIX typedefs and defined constants completely clutter up the name space? How can I be sure I'm not going to use a variable name in my program that's already used? POSIX reserves only a small set of carefully defined symbols:

- all external identifiers that begin with an underscore
- all identifiers beginning with an underscore followed by a capital alphabetic character
- all identifiers ending in _t
- when we include a header file, we reserve all external identifiers placed in the header by Standard C or POSIX.1
- also when we include a header file, we reserve all names containing a special prefix or suffix, such as d_ when we include <dirent.h> or LC_, n_ and p_ when we include <locale.h>.

The most important symbol to be defined is _POSIX_SOURCE, which uses the symbols we've defined and actually requires you to avoid redefining the identifiers in the classes we named above. Furthermore, defining _POSIX_SOURCE requires only a specified list of identifiers to be defined in each header. In other words, if you turn on _POSIX_SOURCE, then the names defined in the header files are mandated by the standard.

So what does this all mean where the programming pencil meets the scratch pad? Let's look at some sample programs.

What does the program in Listing 1 do? Let's start by looking at its output:

Listing 1

```
#define _POSIX_SOURCE

#include <sys/limits.h>
#include <stdio.h>
#include <unistd.h>

main()
{
  printf("_POSIX_NAME_MAX = %d\n", _POSIX_NAME_MAX);
  printf("_PC_NAME_MAX = %d\n", _PC_NAME_MAX);
  printf("pathconf(\".\", _PC_NAME_MAX) = %d\n",
    pathconf(".", _PC_NAME_MAX));
#ifdef NAME_MAX
  printf("NAME_MAX = %d\n", NAME_MAX);
#else
  printf("NAME_MAX not defined\n");
#endif
}
```

```
_POSIX_NAME_MAX = 14
_PC_NAME_MAX = 14
pathconf(".", _PC_NAME_MAX) = 255
NAME_MAX not defined
```

As we've discussed before, _POSIX_NAME_MAX is the smallest maximum file name size, or 14 bytes as defined

Listing 2

```
#define _POSIX_SOURCE
#include <stdio.h>       /* for fprintf()         */
#include <stdlib.h>      /* for EXIT_*, exit()    */
#include <sys/types.h>   /* for stat.h            */
#include <sys/stat.h>    /* for creat(), S_I*     */

#define S_IRW \
      S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH

main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "usage: %s filename\n", argv[0]);
    exit(EXIT_FAILURE);
  }
  if (creat(argv[1], S_IRW) == -1) {
    fprintf(stderr, "creat failed\n");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

in <limits.h>. So the output of the first printf() should be clear.

The routine pathconf() provides information about the system configuration for the file system named in the first argument. What information you request is specified in the second argument.

The _PC_NAME_MAX is a symbolic constant that corresponds to an entry in the table of system limits. (It is mere coincidence that the value of the constant on AIX is the same as the POSIX minimum maximum.) Once we call pathconf(), we find that the longest file name we can have (on the file system where I ran the program) is 255 bytes, which is very long indeed. (Why is the path name an argument? Because I may have a file system remote mounted from a platform where the file name maximum is different. Or I may have different file system types mounted on my computer. In either case, I can have multiple answers to the question on the same system.)

Note that AIX doesn't define a value of _NAME_MAX but requires you to request the maximum file size using pathconf().

Let's try another example.

The touch Command

Generally, when we want to create an empty file, we use the POSIX.2 command touch. How does it work?

We could open the file for reading, but the open would fail since the file doesn't exist. Instead, we use the POSIX interface creat() in the program in Listing 2.

Some points to note: First, we've turned on _POSIX_SOURCE so only the POSIX definitions are going to be included from the header files. Second, we're calling three separate interfaces: fprintf(), which is defined in Standard C; creat(), which is defined in POSIX; and exit(), which returns to the operating system.

We are acting like good programmers, by using header files for the defined constants and function prototypes, including a usage message (though we don't do anything fancy about argument parsing), and checking the return status from the POSIX function. We also exit the program with different values depending on the success of our operation.

What does the constant S_IRW represent? It is composed of the flag values for opening the file with user read (C_IRUSR), user write (C_IWUSR), group read and write (C_IRGRP, C_IWGRP) and other read and write permissions. In other words, create the file with rw-rw-rw permissions.

When we invoke the program on AIX, we get:

```
$ ls -l foo
ls: 0653-341 The file foo does not exist.
$ ./touch foo
$ ls -l foo
-rw-rw-rw-   1 jeff   staff   0 Jul 12 16:58 foo
```

But if we had a file already, we'd have a different result:

```
$ date >foo
$ ls -l foo
-rw-rw-rw-   1 jeff   staff  29 Jul 12 17:00 foo
$ ./touch foo
$ ls -l foo
-rw-rw-rw-   1 jeff   staff   0 Jul 12 17:00 foo
```

What happened to the contents? creat() always truncates the file when it opens it. That's a real problem because the other thing touch is supposed to do is update the access time of a file that already exists, not erase it. So we need to check if the file exists. If it does, we need to open the file. If it doesn't, we can use the same code we used earlier.

The stat Interface

How do we tell if the file exists? We use the stat() interface. stat takes two arguments–a file name and a pointer to a struct stat into which is returned information about the file. What information? Well, take a look at the structure in Listing 3.

Listing 3

```
struct stat
{
   mode_t    st_mode;     /* File mode */
   ino_t     st_ino;      /* File serial number */
   dev_t     st_dev;      /* ID of device containing a directory*/
                          /* entry for this file.  */
   short     st_nlink;    /* Number of links */
   uid_t     st_uid;      /* User ID of the file's owner */
   gid_t     st_gid;      /* Group ID of the file's group */
   off_t     st_size;     /* File size in bytes */
   time_t    st_atime;    /* Time of last access */
   time_t    st_mtime;    /* Time of last data modification */
   time_t    st_ctime;    /* Time of last file status change */
};
```

Notice this definition makes liberal use of the new types we talked about earlier. But be warned: If you look in <sys/stat.h> on AIX, you're going to find a lot more information because the stat block is used by multiple interfaces, which have different requirements. We've only shown the entries we care about now.

Since stat is used to retrieve the information for an ls -l command, the best way to show it in action is to write a small version of ls, as follows:

```
#define _POSIX_SOURCE
#include <stdio.h>      /* for fprintf(), perror()*/
#include <stdlib.h>     /* for EXIT_*, exit()    */
#include <sys/types.h> /* for stat.h           */
#include <sys/stat.h>  /* for stat()           */
main(int argc, char *argv[])
{
 struct stat buf;
 if (argc != 2) {
   fprintf(stderr, "usage: %s filename\n", argv[0]);
   exit(EXIT_FAILURE);
 }
 if (stat(argv[1], &buf) == -1) {
   perror(argv[1]);
   exit(EXIT_FAILURE);
 }
 printf(" %u %u %u %u %u\t%u %u %s\n",
   buf.st_ino, buf.st_mode, buf.st_nlink,
   buf.st_uid, buf.st_gid, buf.st_size,
   buf.st_mtime, argv[1]);
 exit(EXIT_SUCCESS);
}
```

There are three things to note in this program:

- We take only one file name as an argument.
- If we can't access the file, we call perror(), which prints the text corresponding to the last error. Note that we must call perror() directly after the error occurs.
- All the values we printed are unsigned.

So what's the output?

```
$ ./ls foo
 192582    33188 1 208 1   25    742530941 foo
```

This output isn't as informative as we might like, or even what we've come to expect from the standard ls command. So in the next 30 days, consider the following:

- How do we get the mode, uid, gid and time into a more useful form?
- What is the purpose of the fields st_dev, st_atime and st_ctime?
- Why isn't the file name in the stat structure?
- What does this have to do with setting the access time of the file we were talking about when we introduced the stat() interface?

Until next month... ▲

Reader Feedback

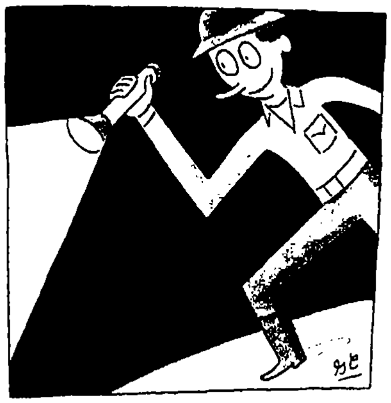
To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
	High	Medium	Low
Features:			
Managing Chaos	170	171	172
IT Hygiene: Clean Layering	173	174	175
Trying on Two X Terminals for Size	176	177	178
Columns:			
Q&AIX–Updates and imake	179	180	181
Systems Wrangler–A Wrangler's Bookshelf	182	183	184
Datagrams–Still More News	185	186	187
AIXtensions–AIX Alternate Authentication	188	189	190
POSIX–Headers, Identifiers and Writing Programs	191	192	193

In Which We Write *ln*

by Jeffreys Copeland and Haemer

If you're just joining us, this is the third column in a series that attacks POSIX from an unconventional angle. Our goal is to help you teach yourself the POSIX system interfaces by thinking about what calls have to exist in order to build the commands you routinely use. Last time, we introduced the `stat()` call to write a simple version of `ls`. Let's take a look at that program again:



```
#define _POSIX_SOURCE
#include <stdio.h>      /* for fprintf(), perror() */
#include <stdlib.h>     /* for EXIT_*, exit() */
#include <sys/types.h>  /* for stat.h */
#include <sys/stat.h>   /* for stat() */

main(int argc, char *argv[])
{
   struct stat buf;

   if (argc !=2){
      fprintf(stderr, "usage: %s filename\n" , argv[0]);
      exit(EXIT_FAILURE);
   }
   if (stat(argv[1], &buf) = = -1) {
      perror(argv[1]);
      exit(EXIT_FAILURE);
   }
   printf(" %u %u %u %u %u\t %u %u %s\n" ,
      buf.st_ino, buf.st_mode, buf.st_nlink,
      buf.st_uid, buf.st_gid, buf.st_size,
      buf.st_mtime, argv[1]);
   exit(EXIT_SUCCESS);
}
```

Jeffrey Copeland (jeff@aus.shl.com) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

When we compile and run it, we get:

```
$ ./a.out foo
192582 33188 1 208 1  25 742530941 foo
```

The output shows that the stat structure (see below) contains virtually all the information about a file that you can get out of ls, even if it's in a cryptic form. In UNIX implementations, all this information is stored in an i-node, one inode per file. In one sense, the inodes are the files: They contain all the file attributes, plus a pointer to the file data. The stat() call is just a convenient way to bundle the attribute information in a tidy package for the programmer.

```
struct stat {
  mode_t st_mode;       /* File mode */
  ino_t st_ino;         /* Inode number */
  dev_t st_dev;         /* ID of device containing */
                        /* a directory entry for this file */
  nlink_t st_nlink;     /* Number of links */
  uid_t st_uid;         /* User ID of the file's owner*/
  gid_t st_gid;         /* Group ID of the file's group */
  off_t st_size;        /* File size in bytes */
  time_t st_atime;      /* Time of last access */
  time_t st_mtime;      /* Time of last data modification */
  time_t st_ctime;      /* Time of last file status change */
                        /* Times measured in seconds since *
                        /* 00:00:00 GMT, Jan. 1, 1970 */
}
```

We also left you with a series of questions. You've had a month to think about them, so let's review them.

• *How do we get the mode, uid, gid and time into a more useful form?*

The mode is easy; it just takes a little translation code. Think of the mode as an octal number, instead of as an unsigned decimal, and it becomes clearer: 33188 becomes 100644. A peek at sys/stat.h tells us that the leading 1 in 100644 means we are looking at a regular file, but the macros and symbolic constants defined in sections 5.6.1.1 and 5.6.1.2 of the POSIX.1 standard relieve us from having to memorize this and let us test it with the S_ISREG() macro. Similarly, we can check to see if the file is a directory with the macro S_ISDIRC().

The last three octal digits, 644, represent the familiar access permission for the owner, group and world. Again, sys/stat.h defines some useful constants for us, shown in the adjacent table. In a rewrite of our first cut at ls, we'd use them to translate 100644 into a more familiar form: -rw-r- -r- -.

Other special routines let us translate the remaining three numbers in our crude ls into more useful forms. The user and group names can be had from the uid and gid values with the routines getgrgid() and getpwuid(). The first takes a gid and returns a group structure as defined in <grp.h> (see Listing 1). Similarly, getpwuid() takes a uid as an argument and returns a passwd structure, from <pwd.h> (see Listing 2).

The time can be converted to a familiar form with the standard C routine localtime(), which provides us with a structure containing (among other things) month, day, year and clock time. Alternately, harking back to our columns on internationalization, we could use the internationalized strftime() function from standard C, which both converts and formats the date in a single function call and can even print the month as a name in your local language.

We encourage you to try using some of these facilities to improve our simple-minded version of ls.

• *What would we use the fields* st_ctime *and* st_atime *for?*

The st_ctime represents the "time of last status change": On UNIX, this is just the inode modification time. st_ctime changes not only when we modify the file itself, but when we change information in the equivalent of the inode–that is, any time we change something we get back from the stat structure. For example, when we do a chmod or chown,

Permissions and POSIX

POSIX name	Traditional UNIX value	Description
S_IRWXU	0700	Permission mask for the owner
S_IRUSR	0400	Owner read
S_IWUSR	0200	Owner write
S_IXUSR	0100	Owner execute or directory search
S_IRWXG	0070	Permission mask for the file group
S_IRGRP	0040	Group read
S_IWGRP	0020	Group write
S_IXGRP	0010	Group execute or directory search
S_IRWXO	0007	Permission mask for others
S_IROTH	0004	Other read
S_IWOTH	0002	Other write
S_IXOTH	0001	Other execute or directory search

Listing 1

```
struct group {
   char * gr_name;     /* group name */
   gid_t gr_gid;       /* group id */
   char ** gr_names;   /* list of group member names */
 };
```

Listing 2

```
struct passwd {
   char *pw_name;     /* user name */
   uid_t pw_uid;      /* user id */
   gid_t pw_gid;      /* primary group id for this user */
   char *pw_dir;      /* initial working directory */
   char *pw_shell;    /* user startup shell */
}
```

Listing 3

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>        /* for strlen() */
#include <sys/limits.h>    /* for _POSIX_PATH_MAX */
extern char *basename(char *);

main(int argc, char *argv[])
{
   char newfile[_POSIX_PATH_MAX];
   struct stat buf;

   if (argc != 3)
        err_quit("usage: %s file1 file2", argv[0]);
   strcpy(newfile, argv[2]);
   if ((stat(argv[2], &buf) != -1)
     && S_ISDIR(buf.st_mode)) {
        strcat(newfile, "/");
        strcat(newfile, basename(argv[1]));
   }
   if (link(argv[1], newfile) == -1)
        err_sys("link failed");
   exit(0);
}
char *basename(char *s)
{
   char *pc;

   for (pc = s + strlen(s); *pc != '/'; pc--)
        if (pc == s)
          return(s);
   return(pc + 1);
}
```

the `st_ctime` changes. You can get the `ctime` out of `ls`, too, with an `ls -lc`. Put another way, `ls -l` tells you the last time you changed the data in a file, while `ls -lc` tells you the last time you changed its attributes. Besides keeping track of the last time you changed a file or its attributes, a POSIX system also keeps track of the last time you fetched data from it. The `st_atime` is the time of the last file access.

Try this experiment:

```
$ ls -lu /etc/passwd
$ cat /etc/passwd
$ ls -lu /etc/passwd
```

This little-known feature can be astonishingly useful. If you have an application that isn't working and you want to know if it's actually reading the input data file, using `ls -lu` is a lot faster than scattering debugging code around your opens and reads. Sadly, the `-c` flag gives you the `ctime`, but the `-a` flag for `ls` was already in use, so you get the `atime` with the `-u` flag. Think of it as the "useful" flag.

• *Why isn't the file name in the stat structure?*

Because the inode doesn't contain a file name. All inodes are kept in an array, analogous to the DOS FAT table, for those of you coming from DOS. If we go back to the earlier statement that each inode represents a file, the file's inode number–the index into this array–is its only real name. We'll soon show you this in more detail. The file names we generally use are, as we shall see, just convenient labels, stored in directories, that point at inodes. The same file can actually have different names in different directories or even more than one name in a single directory.

• *What does this have to do with setting the access time of a file, which led us to the* `stat()` *interface?*

Not much, but it does illustrate some themes that will recur in these columns:

1. UNIX commands are just programs; any system calls they use, you can use, too. The very existence of the `touch` command lets us infer that the operating system stores access time information somewhere and there must be calls that let us get at the information.

2. For most objects and attributes in the system, there are separate calls to get and set information. The creat() we used in touch and the stat() we used in ls are examples of a pattern.

3. Once we've figured out that something exists, looking at other shell-level commands can deepen our understanding. touch shows that file modification times were stored somewhere. Looking at some of the more obscure flags from ls shows us that three times are stored, not just one.

4. Getting there, as they used to say in travel commercials, can be half the fun. We've come all this way and still can't set file modification times except by creating or truncating them. We'll return to setting the access time in a few weeks, but first we'll lay more groundwork by digressing into other things.

The ln program

stat() gets a lot of attributes at once. Our discussion of st_[acm]time shows that setting them is a more piecemeal affair. Let's continue our discussion with the link count, st_nlink. As you know, the ln command allows you to give a file more than one name. What many people don't appreciate (but you do now since you know what is and isn't stored in the stat structure) is that all these names are equivalent–a file's only "real name" is its inode number, which you can get with ls -i. That said, basic system calls like open() expect the same symbolic names we use. So how do we tag a file with a symbolic name? One way is during file creation. A second way, used by ln, is with the system call link().

Listing 3 shows a cut-down version of ln that illustrates the point. We use it like this:

```
$ touch foo
$ mkdir foo.d
$ ./a.out foo fool
$ ./a.out foo foo.d
$ ls -l foo*
-rw-rw-r--  3 jsh other 0 Aug 24 14:27 foo
-rw-rw-r--  3 jsh other 0 Aug 24 14:27 fool

foo.d:
total 0
-rw-rw-r--  3 jsh other 0 Aug 24 14:27 foo
```

Briefly, it does a quick check for proper usage, has a little code for one special case (ln filename directory) and then invokes link(). The code is meant to be read, and we encourage you to do so, but we'll point out a few things:

1. link() requires two filenames. Like the command ln, the call gives the file in the first argument the new name in the second.

2. Wherever possible, we've used symbolic constants and macros supplied by POSIX itself. One example of this is the array newfile[], which holds the new name being attached to the file. Instead of giving it an arbitrary size specific to the program (and the programmer), we set it to _POSIX_PATH_MAX, which is the size of the longest completely portable pathname. The actual maximum path length can vary with the file system, so a more robust version might use pathconf(). Similarly, we test whether the second argument is a directory using the POSIX macro S_ISDIR().

3. Note how the includes have begun to proliferate. POSIX and Standard C are both very clear about what's declared where, which is good, but the lists of include files in our programs will get larger and larger. Sometimes, we may just show the code and omit the includes.

4. The err_sys() and err_quit() routines are not standard. In general, having a handful of routines that print uniformly formatted error messages and then either quit or continue both simplifies your code and ensures that you handle errors consistently. You may already have your own. We've lifted ours from Richard Stevens' excellent book *Advanced Programming in the UNIX Environment* (Addison-Wesley Publishing Co., 1992, ISBN 0-201-5617-7).

Until Next Month...

The ln program has gotten us thinking about file names. Next month, we'll plunge further into this subject by writing mv. In the interim, we'd like to leave you with a few points to ponder, as is our wont. Some of these you can test by typing in the code we've given you and trying it out yourself. That's one reason we try to keep our code samples simple.

• We've done some simple error checking. What sort of error checking should a more robust version have? How would you implement err_*() routines?
• What if argv[1] is a directory? What should happen?
• What if the constructed filenames are too long?
• Even executables can have st_nlink > 1.

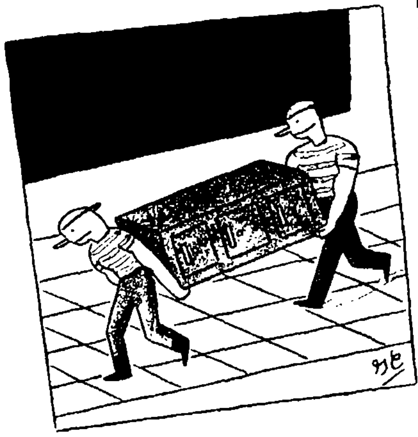
```
$ ls -il /bin/ln /bin/mv /bin/cp
   70 -rwxrwxr-x 3 bin bin 9346 Sep 11 1990 /bin/cp
   70 -rwxrwxr-x 3 bin bin 9346 Sep 11 1990 /bin/ln
   70 -rwxrwxr-x 3 bin bin 9346 Sep 11 1990 /bin/mv
```

When do you want to use this trick, when do you want to avoid it, and if they're really all the same program, why do they behave differently? ▲

In Which We Move and Remove Files

by Jeffreys Copeland and Haemer

Welcome back for the fourth in our series on POSIX. If you've just come in, we're examining the 1003 standard from a different direction: by deducing information about the system interfaces in the applications programming interface of 1003.1 from information we already know about the command interface of 1003.2, which we recognize as almost identical to the UNIX command set with which we are familiar.



Last time, we talked about the ln program and ended up with a version that works when the target is either another file or a directory. We finished the installment with the observation that, on some systems, the programs cp, ln and mv are all the same: They're just links to the same exact executable file. Why is this the case? And in what circumstances is this trick useful for programs of your own? Let's answer those questions last, after we explore the rm and mv commands.

Removing a File

Let's start with the POSIX rm command. How do we remove a file? Well, POSIX provides us with an unlink() interface to remove a link to a file. For example, consider the following program:

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
  if (argc != 2)
    err_quit("usage: %s file", argv[0]);
  if (unlink(argv[1]) == -1)
    err_sys("unlink failed");
  exit(0);
}
```

If we compile it, then run it:

Jeffrey Copeland (jeff@aus.shl.com) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

```
$ ls
macros.mm      posix-2.mm     posix-4.mm
posix-1.mm     posix-3.mm

$ ./a.out posix-4.mm

$ ls
macros.mm      posix-2.mm
posix-1.mm     posix-3.mm
```

Unfortunately, now we have to start writing this article again.

Moving a File

That's removing a file; what does it mean to move a file? Well, on UNIX (or AIX), moving a file is a superset of the DOS rename command. We provide two filenames, and the first file is replaced by the second. For example:

```
$ ls
macros.mm      posix-2.mm     posix-4.mm
posix-1.mm     posix-3.mm

$ mv macros.mm foo.bar

$ ls
foo.bar        posix-2.mm     posix-4.mm
posix-1.mm     posix-3.mm
```

But, unlike DOS' rename (which one of us is being forced to use a lot lately, much to his frustration), mv allows you to move a file elsewhere, viz.,

```
$ mkdir x

$ ls -R
.:
macros.mm      posix-2.mm     posix-4.mm
posix-1.mm     posix-3.mm     x

./x:

$ mv macros.mm x

$ ls -R
.:
posix-1.mm     posix-3.mm     x
posix-2.mm     posix-4.mm

./x:
macros.mm
```

You can also move a file to a different name in a different directory by doing something like:

```
$ mv macros.mm /tmp/foobar
$ ls /tmp
_ndxch      bedaaidm      foobar      mf0228.tmp
```

(Notice that you don't end up with an annoying `Invalid filename or file not found` message when you try this.)

From our version of ln last month, can you guess how mv actually works? Go ahead and think about it; we'll grab a beer while you do. (Both Austin and Boulder, by the way, have several excellent microbreweries. The one we favor in Austin has a neat tour, but the ones in Boulder serve lunch.)

All set? As you no doubt figured out, it's pretty simple: You create a link to the new name and then use unlink() to remove the old name, like this:

```
main(int argc, char *argv[])
{
  char newfile[_POSIX_PATH_MAX];
  struct stat buf;

  if (argc != 3)
   err_quit("usage: %s file1 file2 , argv[0]);
  strcpy(newfile, argv[2]);
  if ((stat(argv[2], &buf) != -1)
   && S_ISDIR(buf.st_mode)) {
    strcat(newfile, "/");
    strcat(newfile, basename(argv[1]));
  }
  if (link(argv[1], newfile) == -1)
   err_sys("link");
  if (strcmp(basename(argv[0]), "mv") == 0)
   if (unlink(argv[1]) == -1)
    err_sys("unlink");
  exit(0);
}
```

We need to note that the file itself isn't changed by this program. We didn't change the inode at all, just the name of the file. An interesting point is that this program now does both ln and mv: If we invoke it as anything other than mv, it doesn't do the unlink. Notice that we used last month's basename(), err_quit() and err_sys() routines all over again. Also notice that both the err routines, which print an error message and exit, help us avoid really messy program structure. (err_quit prints its argument and exits; err_sys also prints the system's errno value.) For example, if they only printed error messages, we would end up with code like this:

```
if (argc != 3)
 err_quit("usage: %s file1 file2 , argv[0]);
else {
 /* ... */
```

Figure 1

```
$ ls -li .
total 140
   6174    -rw-r-r-           1 jeff       other       1898    Sep 11 23:00   coding-notes
    386    -rw-r-r-           1 jeff       other        417    Sep 11 23:00   invoice
   8989    -r-xr-xr-x         1 jeff       other       1805    Sep 11 23:00   macros.mm
    390    -rw-r-r-           1 jeff       other        555    Sep 11 23:00   makefile
   9854    -rw-r-r-           1 jeff       other       1715    Sep 11 23:00   p3-corr
    397    -rw-r-r-           1 jeff       other      21745    Sep 11 23:00   posix-1.mm
    399    -r-r-r-            1 jeff       other      13934    Sep 11 23:00   posix-2.mm
    403    -r-r-r-            1 jeff       other      13453    Sep 11 23:00   posix-3.mm
    406    -rw-r-r-           1 jeff       other       7431    Sep 11 23:00   posix-4.mm
    407    -rwxr-xr-x         1 jeff       other        745    Sep 11 23:00   strip

$ od -cd .
0000000   06155    00046    00000    00000    00000    00000    00000    00000
         013 030   .
0000020   06323    11822    00000    00000    00000    00000    00000    00000
         263 030   .  .
0000040   06174    28515    26980    26478    28205    29807    29541    00000
         036 030   c   o    d   i    n   g    -   n    o   t    e   s
0000060   00386    28265    28534    25449    00101    00000    00000    00000
         202 001   i   n    v   o    i   c    e
0000100   08989    24941    29283    29551    27950    00109    00000    00000
         035   #   m   a    c   r    o   s    .   m    m
0000120   00390    24941    25963    26982    25964    00000    00000    00000
         206 001   m   a    k   e    f   i    l   e
0000140   09854    13168    25389    29295    00114    00000    00000    00000
           ~   &   p   3    -   c    o   r    r
0000160   00397    28528    26995    11640    11825    28013    00000    00000
         215 001   p   o    s   i    x   -    1   .    m   m
0000200   00399    28528    26995    11640    11826    28013    00000    00000
         217 001   p   o    s   i    x   -    2   .    m   m
0000220   00403    28528    26995    11640    11827    28013    00000    00000
         223 001   p   o    s   i    x   -    3   .    m   m
0000240   00406    28528    26995    11640    11828    28013    00000    00000
         226 001   p   o    s   i    x   -    4   .    m   m
0000260   00407    29811    26994    00112    00000    00000    00000    00000
         227 001   s   t    r   i    p
0000300
```

(Did we just do a dump of the directory? Yes. It's just a file. Notice the numbers: they're the inodes.)

```
$ rm coding-notes

$ od -cd .
0000000   06155    00046    00000    00000    00000    00000    00000    00000
         013 030   .
0000020   06323    11822    00000    00000    00000    00000    00000    00000
         263 030   .  .
0000040   00000    28515    26980    26478    28205    29807    29541    00000
                   c   o    d   i    n   g    -   n    o   t    e   s
0000060   00386    28265    28534    25449    00101    00000    00000    00000
         202 001   i   n    v   o    i   c    e
0000100   08989    24941    29283    29551    27950    00109    00000    00000
         035   #   m   a    c   r    o   s    .   m    m
0000120   00390    24941    25963    26982    25964    00000    00000    00000
         206 001   m   a    k   e    f   i    l   e
0000140   09854    13168    25389    29295    00114    00000    00000    00000
           ~   &   p   3    -   c    o   r    r
0000160   00397    28528    26995    11640    11825    28013    00000    00000
         215 001   p   o    s   i    x   -    1   .    m   m
0000200   00399    28528    26995    11640    11826    28013    00000    00000
         217 001   p   o    s   i    x   -    2   .    m   m
0000220   00403    28528    26995    11640    11827    28013    00000    00000
         223 001   p   o    s   i    x   -    3   .    m   m
0000240   00406    28528    26995    11640    11828    28013    00000    00000
         226 001   p   o    s   i    x   -    4   .    m   m
0000260   00407    29811    26994    00112    00000    00000    00000    00000
         227 001   s   t    r   i    p
0000300
```

```
 if (link(argv[1], newfile) == -1)
   err_sys("link");
 else if (strcmp(basename(argv[0]), "mv") == 0) {
    if (unlink(argv[1]) == -1) {
      err_sys("unlink");
    else
      exit(0);
  else
    exit(0);
 }
}
exit(1);
```

Such error-routine interfaces aren't in any of the standards, but it's useful to have a set of your own and straightforward to create them. As we noted last time, we're using the ones from Richard Stevens' *Advanced Programming in the UNIX Environment* because we like their design. Besides, they're well-thought-out and already debugged. Why reinvent the wheel? Now, the inevitable points to ponder about the preceding code:

- What happens if the file already exists? The link fails, and the program aborts.
- If `unlink()` only removes a name, where's it removing the name from? The directory. Our `link()/unlink()` combination adds a directory entry for the new name and then deletes the directory entry for the old one. As we said before, we never touch the contents of the file. We update the `st_ctime` of the file's inode, and the `st_ctime` and `st_mtime` of the directories containing the names.
- Well, then what's in a directory? Patience, we'll come to that soon, but we've still got an outstanding question from last month.

In order to answer the outstanding question, we need to make an observation first: Why does the following happen?

```
$ cat /etc/passwd >a

$ ls -l a
-rw-r--r-- 1 jeff   staff  1671 Sep 10 10:54 a

$ ./mv a b          # our own version of mv

$ ls -l a b
ls: 0653-341 The file a does not exist.
-rw-r--r-- 1 jeff   staff  1671 Sep 10 10:54 b

$ pwd
/user/jeff/posix

$ ./mv b /tmp
sys_err: link
```

The purple-and-yellow book (POSIX.1) tells us that `link()` can result in the error EXDEV, when the location given by the new name and the existing file are on different file systems, and the implementation doesn't support links between file systems. Most existing UNIX systems–including AIX–have this implementation restriction. When we tried to create the link from `/user/jeff/posix/b` to `/tmp/b`, we were crossing a file system boundary, and the link failed.

How do we get around this? Our implementation of `mv` is pretty useless if we can't move files across file systems. If we are on a different file system, we have to open the new file, copy the contents of the old file to the new file, and then unlink the old file. So `mv` ends up having almost all of `cp` in it. And, as we've noticed, `ln` is a subset of the code in `mv`. So why not just build all three of them as the same program, and then have slightly different execution behavior based on the name of the file? That, gentle reader, is why the following is the case on some UNIX systems.

```
$ ls -il /bin/ln /bin/mv /bin/cp
 70  -rwxrwxr-x  3 bin  bin  9346  Sep 11 1990   /bin/cp
 70  -rwxrwxr-x  3 bin  bin  9346  Sep 11 1990   /bin/ln
 70  -rwxrwxr-x  3 bin  bin  9346  Sep 11 1990   /bin/mv
```

This trick simplifies code maintenance, but you should avoid using it unless the amount of code overlap is really large–otherwise, you're just packing a lot of independent, unrelated functions into a single executable. (Those of you old enough to have used CP/M will remember `pip`, which we still suspect could have made our microcomputers dispense root beer if only we could have found the right options.)

Directories

As we've probably mentioned before, on a UNIX system a file is a file is a file. This means that we can treat any file the same. For example, in Figure 1 we delete the file, and the inode changes to zero.

Thoughts for Next Time

Next time, we'll explore directories more thoroughly. In preparation, we'll close with a few things for you to try yourself and a few things for you to scratch your head about late at night when you can't sleep.

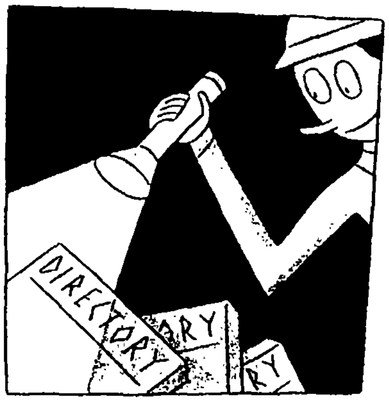
- What happens if I create a new file in the directory shown above? In what order will the entries of that directory print if I do an `ls`?
- If `unlink()` only removes a directory listing and doesn't do away with a file, how *do* we get rid of it?
- Why is there a `lost+found` directory in each file system, and why is it so big? (Try `ls -ld /lost+found` and contrast the size with that of an empty directory that you create yourself.) How might you write `ls`? ▲

In Which We Explore Directories

by Jeffreys Copeland and Haemer

In our last column, we began discussing directory structure. This time, we'll begin by exploring POSIX directories in more detail. We'll move on to rewriting our toy ls program from several columns ago. We'll finish by discussing file modes, thereby setting up a discussion of the files next month.

To review from last time, we observed that a directory is just a file. If we do an octal dump of a directory on AIX, we find it contains a number of entries of 16 bytes each. The first two bytes appear to be hieroglyphics; the last 14 are the first bytes of the file name. For example:



```
$ od -cd .
 .
 .
0000320 360 350   p    o    s    i    x    -    4    .    m    m
             61672  28783    29545    30765   13358    28013   00000 00000
0000400
```

As we discovered, the two bytes of hieroglyphics are actually the inode number.

```
$ ls -l posix-4.mm
 61672 -r--r--r--   1 jeff    staff    13661 Sep 13 13:49 posix-4.mm
```

In a full-page demonstration, we found that the inode number is reset to zero on some POSIX systems when we delete the file from the directory. This leads us to the questions we left you with last time:

Jeffrey Copeland (jeff@aus.shl.com) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*
Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

• *What happens if I create a new file in the directory where we've deleted a file?*

When we create a file, it's inserted into the first available space in the directory. If we've deleted a file, it replaces the entry with the zero inode. If we have no empty spots available, it is added onto the end.

• *In what order will the entries of that directory print if I do an* `ls`*?*

They magically appear in alphabetical order. (How does that happen? Hint: See `qsort` in the C Standard.)

• *If* `unlink()` *only removes a directory listing and doesn't actually do away with a file, how do we get rid of it?*

If there are additional links to the file, nothing happens to the data itself. When the link count in the inode goes to zero, the system frees the blocks, and they become available for use by another file.

• *Why is there a* `lost+found` *directory in each file system, and why is it so big?*

Traditionally, UNIX doesn't write its files in real time. Sometimes, there are files on the disk that have not yet been attached to directories. There are files that have been deleted from directories but haven't been purged from the disk. When the system crashes, we need to do an integrity check of the file system to catch the files in these in-between states. This is the purpose of the program `fsck`. The `lost+found` directory is the catch-all for these loose files. Why is it so big? Because when the system is starting up in single-user mode, `fsck` is operating on the raw device and doesn't deal with the directory through polite system calls. We make its life easier by giving it a premade directory with empty slots into which it places loose files it finds; since it can't increase in size, we need to supply it beforehand with a number of empty slots. In the dark ages, when a file system was created, the system manager did something like:

```
cd /newfilesystem
mkdir lost+found
cd lost+found
for i in 1 2 3 4 5 6 7 8 9 ...
do
  for j in 1 2 3 4 5 6 7 8 9 ...
  do
  touch $i$j
done
rm *
```

which left the file system big enough to hold a lot of files. Two things to notice about this trick: Since directories never shrink in size, `lost + found` can always hold at least the number of files it was set up for. Also, `fsck` is only looking for a place to put the directory entry–the file already exists, and already has its space on disk.

• *How might you write* `ls`*?*

The obvious way of doing an `fopen()` on the file ".", reading information from it in 16-byte chunks, sorting it by file name, and printing it out would work. But it's wrong. How do you know that the format of the directory entry will be the same on the next POSIX-compatible system you work on? POSIX tells us nothing about the format of the directory entries. It is only tradition that has them laid out as in our examples. Indeed, on AIX, the "inode number" in the directory does not necessarily even correspond to the inode number reported by `stat()`, or shown in an `ls -li`.

The Real ls Program

Fortunately, POSIX provides us with some handy routines for getting at directories, and, of course, a data structure for handling information from them. We can begin by looking at the POSIX header file `<dirent.h>`. The POSIX standard tells us that we have a `DIR`, which is analogous to the `stdio FILE` type for directories, and a `struct dirent` containing at least the array `char d_name[]`, which is no more than `NAME_MAX+1` bytes long. AIX is a little more generous with its information: We also get the inode number (d_ino) and the length of the file name (d_namelen) in `dirent`. In a fully POSIX-portable program you would get this information by calling `stat()` with the file name.

To access the directory structures, we have four convenient routines:

- `opendir()` takes the name of the directory and returns a pointer to a `DIR`,
- `dirent()` returns a pointer to the next `dirent`,
- `closedir()` closes the open directory, and
- `rewinddir()` allows us to restart at the beginning of the directory we're reading.

For example, we can list the files in the current directory with the following program:

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

main()
{
  DIR *dirp;
  struct dirent *dp;

  if( (dirp = opendir( "." )) == NULL )
     err_quit( "opendir failed" );
  while( (dp = readdir(dirp)) != NULL )
     puts( dp->d_name );
  closedir( dirp );
  exit( 0 );
}
```

which provides output like

```
.
..
macros.mm
makefile
posix-1.mm
posix-2.mm
RCS
foo.c
posix-3.mm
posix-4.mm
posix-5.mm
foo
```

Not quite the `ls` you're used to, is it? Several things to notice: First, both the current directory (`.`) and its parent (`..`) are listed in the output. You'd expect this, since you're used to seeing the way UNIX directories are linked together, when you use `ls -a`. Second, the entries are not in any apparent order. Third, how would we do a listing of a different directory? Lastly, we are only seeing the file names. How would we do an `ls -l`? The first problem is solved with a simple change:

```
while( (dp = readdir(dirp)) != NULL )
   if( *dp->d_name != '.' )
     puts( dp->d_name );
```

The remaining three problems we will leave as an exercise for the reader. Note that we already did a large part of solving the last problem in an earlier column.

File Modes

That leads us neatly to the question of file modes. When you do an `ls -l`, you get some information about the file attributes, viz.,

```
-rw-r--r-- 1 jeff  staff  6824 Oct 07 16:04 posix-5.mm
```

How do they get set? How do we modify them? As we've discussed before, POSIX defines symbolic constants for each of the traditional UNIX protection bits. For example, S_IXGRP represents the execute permission bit for group, and S_IWUSR is the user write bit. We already know about the `chmod` command.

```
$ touch bar
$ ls -l bar
-rw-r--r--     1 jeff      staff      0 Oct 07 23:11 bar
$ chmod 777 bar
$ ls -l bar
-rwxrwxrwx     1 jeff      staff      0 Oct 07 23:11 bar
$ chmod 3 bar
$ ls -l bar
--------wx     1 jeff      staff      0 Oct 07 23:11 bar
```

As you might surmise, there's an underlying interface to do the dirty work. We'll use that interface, conveniently named `chmod()`, to build a version of the command:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
int mode_bit[12] = {
  S_IXOTH, S_IWOTH, S_IROTH,
```

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
Features:	High	Medium	Low
Network Management as Team Sport	170	171	172
Software Soup	173	174	175
The Cadillac of Clusters	176	177	178
Columns:			
Q&AIX–Performance Monitoring	179	180	181
Systems Wrangler–Look What They've Done to NFS	182	183	184
Datagrams–Back to the Future	185	186	187
AIXtensions–Paint by Numbers, the RS/6000 Way	188	189	190
POSIX–In Which We Explore Directories	191	192	193

```
    S_IXGRP, S_IWGRP, S_IRGRP,
    S_IXUSR, S_IWUSR, S_IRUSR,
          0, S_ISGID, S_ISUID,
};

main(int argc, char *argv[])
{
    mode_t mode = 0;
    int n;
    int i;

    if ((argc != 3) || !isdigit(*argv[1]))
        err_quit("usage: %s octal-mode file",
            argv[0]);
    n = strtol(argv[1], NULL, 8);
    for (i=0; i<12; i++)
        if (n & (1<<i))
            mode = mode | mode_bit[i];
    if (chmod(argv[2], mode) == -1)
        err_sys("chmod failed");
    exit(0);
}
```

Things You Should Notice About This Program

• File permissions are type mode_t not type int any more. At the same time, we now use the symbolic constants to refer to the mode bits, not the familiar UNIX numeric modes, such as 0644. In other words, under POSIX, the mode bits are not guaranteed to map neatly to octal constants.

• In POSIX.2, we can still accept the old arguments, such as

```
chmod 644 foobar
```

but this is considered obsolete. Why? Because we may be implementing POSIX commands on a system that doesn't support file permissions in quite the same way as the original Kernighan patent for UNIX mode bits specified. In this case, neither the symbolic constants (such as S_IRWXU), nor the numeric values we'd use on the command line (such as 0711), would match the underlying operating system.

• Two programming tips are notable here: First, by using a table lookup, we made our code a great deal easier. (We've spent some of our days lately trying to fix a 100,000-line program that consistently uses switch statements instead of tables. This makes the code a great deal more difficult to understand, and at least six times bulkier.) Second, using strtol() to convert the command-line argument to a long is exactly the right thing. In particular, by invoking the program as

```
chmod 640 foobar
```

strtol() automatically takes the 640 as an octal since we use a base argument of 8.

What would the code do if we invoked the program as

```
chmod 0777qz foo
```

How can you fix it to catch the error?

• We're also allowed to use command arguments in a newer form, such as

```
chmod g+xs,u-r,o=r foo
```

How much more difficult would it be to parse these? And how would you go about it? Is it enough of a hint to note that the POSIX.2 standard provides a BNF grammar for the command line to chmod?

If We Can Change Permissions, How About Ownership?

You would think that we should be able to modify all of the file attributes we see when we do an ls -l. We can. For example:

```
$ touch bar
$ ls -l bar
-rw-rw-rw-    1 jeff      staff      0 Oct 07 23:26 bar
$ chown 0 bar
$ ls -l bar
-rw-rw-rw-    1 root      staff      0 Oct 07 23:26 bar
$ rm -f bar
$ touch bar
$ chown bin bar
$ ls -l bar
-rw-rw-rw-    1 bin       staff      0 Oct 07 23:26 bar
```

As you can probably guess by now, the chown program uses an underlying chown() interface. We can build it into a program like so:

```
#include <pwd.h> /* for getpwuid */
/* ... other includes */

main(int argc, char *argv[])
{
    struct stat buf;
    uid_t uid;
    struct passwd *pwd;
    char *cp;
    if (argc != 3)
        err_quit("usage: %s uid file , argv[0]);
    if (stat(argv[2], &buf) == -1)
        err_sys("stat failed");
    for(cp = argv[1]; *cp && !isdigit(*cp); cp++)
        ;
    if (cp == '\0')
       uid = (uid_t) atol(argv[1]);
    else {
```

```
    if ((pwd = getpwnam(argv[1])) == NULL)
      err_quit("unknown user id %s",
        argv[1]);
    uid = pwd->pw_uid;
  }
  if (chown(argv[2], uid, buf.st_gid) == -1)
    err_sys("chown failed");
  exit(0);
}
```

For reference, getpwnam() returns a password structure:

```
struct passwd {
  char *pw_name;  /* User name */
  uid_t pw_uid;   /* User ID number */
  gid_t pw_gid;   /* Group ID number */
  char *pw_dir;   /* Initial Working Directory */
  char *pw_shell; /* Initial User Program */
};
```

We can have other fields in passwd in addition to the POSIX-mandated minimum set. For example, char *pw_passwd might contain the actual text of the hashed password. The text of the password is probably not available on systems where there is a file /etc/shadow to contain the actual password data. (An aside: Several years ago, shortly after the Internet Worm disaster, one of us did the quick experiment of testing the passwords on all the UNIX machines he could get to on the local-area net at the company where he worked. He only used a dictionary attack. The only passwords he successfully broke this way belonged to corporate officers. If he had been less scrupulous, he would have given himself a large raise and retired. The moral of the story is that easily guessed passwords really *are* a danger, if not from an external intruder, from the cracker across the hall.) We're just about out of space, so we'll leave you (as usual) with a few points to consider in the next month:

• Why don't we declare the uid and gid to be int instead of having special types uid_t and gid_t ?

• Life would certainly be easier if we had the inverse of stat(), say:

```
setstat( char *filename, statbuf *buf );
```

Why doesn't it exist?

• Since the chown() interface sets both owner and the group, should we just make chgrp a link to chown like we described for mv, cp and ln?

Next month we'll explore file types, file contents and file time stamps. Until then! ▲

WE BUY, SELL & RENT

RS/6000

PARTS • FEATURES
UPGRADES • CPU'S

• System 36 Conversions
• AutoCad Available for RS/6000

New & Used IN STOCK
Complete Technical Center, Installation, Stock Parts & Features for RISC.

COMPUTER® MARKETPLACE

(800) 858-1144 Ext. 95

A Publicly Traded Company NASDAQ:

CDLA IBM® asa

205 East 5th St., Corona, CA 91719
TEL (909) 735-2102 • FAX (909) 735-5717

Circle No. 3 on Inquiry Card

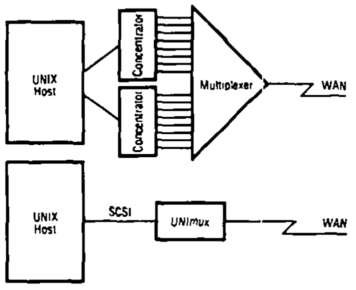
NEW FROM DCB

CLEAN UP CABLE CLUTTER —AND REDUCE YOUR COSTS— WITH THE NEW DCB UNIMUX

Eliminate that snarl of wires to your standard multiplexer. With the DCB UNImux, you can make *all* your local connections with a single cable to a SCSI interface and reduce the number of slots used.

Replace this cable mess:

With the UNImux solution:



For AIX, HP-UX, SCO, DEC and SUN.

Data Comm for Business, Inc.

Manufacturers of modems, multiplexers, everything for your data communications application.

807 Pioneer, Champaign, IL 61820

Toll-free: (800) 637-1127

In Illinois: (217) 352-3207

Circle No. 5 on Inquiry Card

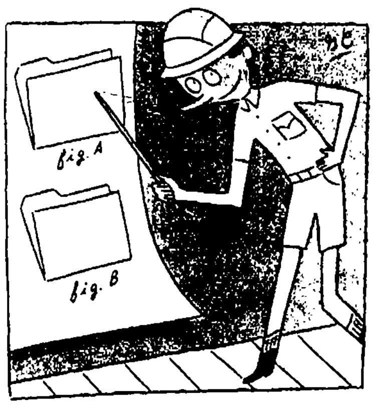
In Which We Discuss File Attributes

by Jeffreys Copeland and Haemer

In case you're just joining us, this column is for you if you're a C programmer who's interested in the POSIX.1 system interfaces. Rather than run through a dry list of system calls, we've chosen to ask what system calls need to exist in order to support the day-to-day commands we use from the command line. Instead of touring the standard, our not-very-hidden goal is to get you to the point where you can tour the standard yourself. Our general approach has to been to make each column a sandwich: a discussion of some common commands and their underlying system facilities, laid between a set of leading questions and the answers to the questions we raised in the preceding column. We like that, so we'll stick with it. First, last month's questions.

When Last We Left our Story...

Last month, we were exploring the file attributes provided by `stat()`. Even though POSIX steers well clear of implementation details–there are POSIX-conforming systems that



aren't UNIX–real UNIX systems, like AIX, store such attributes in inodes. Some questions that we trotted out during those explorations, and our answers:

• *Why don't we declare the uid to be an* `int` *instead of having the special type* `uid_t`*?*

Well, how big is an `int`? On an RS/6000, `INT_MAX` is 2,147,483,647. Seems like that would be big enough, eh? Perhaps not. First, it's smaller than the world population (which is now above 5 billion). Second, not every user is a person. Once

Jeffrey Copeland (`jeff@aus.shl.com`) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*
Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

we admit the possibility that every device in the world that speaks to a computer might want one or more unique uids, an int looks smaller and smaller. Alternatively, we might want uid_t to be smaller than an int. Many traditional UNIX implementations, like ULTRIX, use a short int–15 or 16 bits–to hold a uid_t. And though space grows cheaper all the time, it's never free; we have even used UNIX implementations on which ints are smaller than 32 bits. Computing changes quickly, but standards are crafted to be stable and slow to change. (As the Romans used to say: "Ars longa, machine-architecture-generation-time brevis.") With this in mind, POSIX generally takes the approach of giving quantities of unknown size their own types.

• *Life would certainly be easier if we had the inverse of* stat(). *Why doesn't it exist?*

```
setstat(char *filename, statbuf *buf)
```

Here, we hide behind that most technical of answers: "We don't know." We raise the question neither to advertise our ignorance nor to argue the rationality of the design of the interfaces, but to highlight an important point: POSIX isn't supposed to design "better" versions of what's out there, it is supposed to standardize existing practice. Existing practice gets all the file attributes with a single call, stat(), but sets them individually, with calls like chown().

• *Since the* chown() *interface sets both owner and the group, should we just make the commands* chgrp *and* chown *links to the same command?*

On some systems, they are.

Onwards and Upwards

It is sometimes said that UNIX has no file types. The distinction being made is that older operating systems often provide a wealth of file types for storing data. On such systems, applications programmers must choose the type of file to store their data in, and such choices affect what the operating system subsequently lets you do with the data. Because UNIX lacks such pervasive, OS-level file-typing, most programs can be file-type independent–cat can cat any file–with an enormous, attendant simplification of development, use and maintenance. That said, although UNIX data files are all just byte streams, UNIX really does have a few file types. Figure 1 illustrates this with some simple directory listings.

The left-hand character in a long (ls -l) listing says whether the file is a regular file (-), a directory (d), a character special file (c), a block special file (b), or a fifo (p). Additionally, some files are executable, as shown by the character x in the permissions flags. (Setuid programs or setgid programs show an s instead of an x in the appropriate position.)

Note that mkfifo is the same as old mknod *filename* p. It creates a file in the file system that operates just like a pipe; data come out in the same order they're put in, and reading data from a fifo also removes the data.

You should be able to infer by now that POSIX lets you get at file-type information and display it. If it didn't, how could anyone have written ls? Watch what happens in Listing 1.

Once again, POSIX provides symbolic constants and macros to do what used to be done with bit-fiddling. Read, write and execute permissions are determined with the masks S_I(RWX)*; setuid and setgid behavior are detected with S_ISUID and S_ISGID; the modes themselves are tested with a suite of S_IS*() macros. All operate on the st_mode field of the stat structure that we discussed in an earlier column. Let's pause for a moment to note some general programming points:

• We start with the general case (char permstr[10] ="?rwxrwxrwx";) and loop through an array of masks, blanking out anything that doesn't fit. This saves us from having to write a very large set of nested ifs.

• Notice that a stat() failure doesn't cause the program to exit. We make this point because we've seen blanket statements, in print, that a program should always abort when a system call fails.

• All multiway switches should have a "can't happen" case. We steal this good advice from *Software Tools*, by Kernighan and Plauger, (Addison-Wesley Publishing Co., 1976, ISBN 0-201-03669-X), a must-read book for the UNIX/C programmer even though its code is superfluous on UNIX systems and isn't in C. By having one in ours, we make our code

Figure 1

```
$ ls -l /etc/passwd
-r--r--r--     1 root  sys     1017   Aug 20  09:08   /etc/passwd
$ ls -ld /dev
drwxrwxr-x     7 root  sys     2736   Sep 16  1991    /dev
$ ls -l /dev/console
crw--w--w-     4 root  other   5, 0   Aug 25  17:04   /dev/console
$ ls -l /dev/hd01
brw-------     1 root  sys     0, 3   Mar 28  1990    /dev/hd01
$ mkfifo /tmp/FIFO
$ ls -l /tmp/FIFO
prw-rw-r--     1 jsh   other   0      Aug 26  11:02   /tmp/FIFO
$ ls -l /bin/mail
-rwxr-sr-x     2 bin   mail    44416  Feb 2   1990    /bin/mail
$ ls -l /bin/df
-rwsr-xr-x     2 root  bin     10536  Feb 21  1990    /bin/df
```

portable to systems that have more file types than those required by POSIX.1-1990. Like what? Take a look at sys/stat.h, search for S_ISFIFO, and see what else is on *your* machine.

• Speaking of guarding against later developments, always use a comma after the last element of an array initializer.

Listing 1

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/stat.h> int mode[10] = {
        0,
        S_IRUSR, S_IWUSR, S_IXUSR,
        S_IRGRP, S_IWGRP, S_IXGRP,
        S_IROTH, S_IWOTH, S_IXOTH,
};
char permstr[10] = "?rwxrwxrwx";

main(int argc, char *argv[])
{
        while(--argc)
                doperms(*++argv);
        exit(0);
}
doperms(char *filename)
{
        struct stat buf;
        int i;
        char perms[10];

        if (stat(filename, &buf) = = -1) {
                err_ret("%s", filename);
                return;
        }
        strcpy(perms, permstr);
        for (i=1; i<10; i++)
                if (!(buf.st_mode & mode[i]))
                        perms[i] = '-';
        if ((S_IXUSR & buf.st_mode)
          && (S_ISUID & buf.st_mode))
                perms[3] = 's';
        if ((S_IXGRP & buf.st_mode)
          && (S_ISGID & buf.st_mode))
                perms[6] = 's';
        if (S_ISDIR(buf.st_mode))
                perms[0] = 'd';
        else if (S_ISCHR(buf.st_mode))
                perms[0] = 'c';
        else if (S_ISBLK(buf.st_mode))
                perms[0] = 'b';
        else if (S_ISREG(buf.st_mode))
                perms[0] = '-';
        else if (S_ISFIFO(buf.st_mode))
                perms[0] = 'p';
        else
                err_msg("Non-standard file type %s",
                        filename);
        printf("%s , perms);
}
```

```
int mode[10] = {
    0,
    S_IRUSR, S_IWUSR, S_IXUSR,
    S_IRGRP, S_IWGRP, S_IXGRP,
    S_IROTH, S_IWOTH, S_IXOTH,
};
```

This lets you update the code more easily when you port to a system with more modes than the required POSIX set.

Time

"Time, time, time, is on my side,
Oh yes it is."
–M. Jagger and K. Richards

Or, in our case, "Oh yes they are." It will not have escaped the notice of the discerning reader that ls -l gives one time, but the stat structure contains three. What the heck are these other times? Some experiments are useful here. After a peek at the ls man page, which reveals that we can probe the three times with ls -l (st_mtime), ls -ul (st_atime), and ls -cl (st_ctime), we try some manipulations in Listing 2.

When the file is first created, by the first touch, all three times are the same. The next two experiments, with touch -m and touch -a, show that the three quantities can be manipulated independently. The last two experiments show that, although modifying the contents of the file changes all three quantities, just looking at the file changes the st_atime field (the "a" stands for "access"). Indeed, ls -ul is an old programmer's trick (and one of us, on the verge of a birthday, feels particularly old this evening). Remember the last time you were developing or porting a program and you went in to put printf() calls around the open(), in frustration, to see whether you were even reading the input data file? As you can now see, an ls -ul inputfile is a lot easier. We use the mnemonic "-u for 'useful.'" The st_ctime flag changes whenever the contents of the stat structure change (yes, this is circular, and no, there's no positive feedback loop). This last time stamp is less often useful for the programmer, but still worth knowing. OK, the times are separate and comprehensible, and an application can find each with a `stat()` call. How can an application modify them? With `utime()`. Listing 3 shows an example–a more sophisticated version of the `touch` command we wrote in an earlier column.

Some noteworthy POSIX points about this program:

• `utime()` lets programmers set `st_mtime` and `st_atime`; `st_ctime` is set as a byproduct because `utime()` modifies the inode. (POSIX doesn't say *inode*, but that's how to think about it.)

• `time()` returns seconds since "the epoch" (January 1, 1970). It's depressing to realize that we were born before time began. As usual, POSIX defines a new data type for times, `time_t`, to guard against the inevitability that any standard C data type we might choose would eventually be wrong. (Dave Taenzer often reminds us that even 50-50 odds are 10-to-1 against you.)

• access(*filename*, 0) asks whether a file exists. In general, `access()` lets you ask about your real access permissions to a file. This call is particularly useful in setuid programs, where you might be running as root but want to ask about what you could do to the file if you weren't. As a special case, a second argument of 0 just asks, "Does the file exist?" We could do a `stat` call, but we rather like this special case, and it seemed like a good excuse to use it.

In addition, there are a few general programming points worth making:

• We've restricted `main()` to argument handling. Everything else is a function call. Not only does one of us (JSH) frequently use this as a convention, it's one that he didn't invent. In other words, you may see others use it, so we present it as a public service.

A couple of other common organizational conventions that we use here are also noteworthy. The first is that we use globals to record flag settings. If you find a -z flag during argument parsing, chances are very, very good that the code will immediately set the global variable "zflag". The second is that the code first processes flags, which are marked by a leading "-", and then loops through the remaining arguments assuming they're filenames.

• We've defined a no-argument macro, `USAGE()`. We could have defined it without parentheses, as

```
#define USAGE   \
 err_quit("usage: touch [-am] file", argv[0])
```

but we find the parens helpful. One of the following macros is a symbolic constant and one is a macro evaluated at run time: `MB_CHAR_MAX`, `MB_CUR_MAX`. Quick–which is which?

Will She Be Rescued in the Nick of Time?

Tune in next month and find out. While you're waiting for the next installment in our serial, think about a few things that we've raised in this one.

• If we typedef `time_t` to be a signed, 32-bit quantity, when will time end?

• When should commands take multiple filenames as arguments? No filenames as arguments?

• The POSIX.2 `touch` command has the following one-line synopsis

```
usage: touch [-amc][-r file|-t
[[CC]YY]MMMDDhhmm[.SS]] file ...
```

Listing 2

```
$ ls foo
foo: No such file or directory
$ touch foo
$ ls -l foo; ls -ul foo; ls -cl foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:39 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:39 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:39 foo
$ sleep 60
$ touch -m foo
$ ls -l foo; ls -ul foo; ls -cl foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:41 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:39 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:41 foo
$ sleep 60
$ touch -a foo
$ ls -l foo; ls -ul foo; ls -cl foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:41 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:42 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:42 foo
$ sleep 60
$ echo "hello, world" > foo; touch foo
$ ls -l foo; ls -ul foo; ls -cl foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:43 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:43 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:43 foo
$ sleep 60
$ cat foo > /dev/null
$ ls -l foo; ls -ul foo; ls -cl foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:43 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:44 foo
-rw-rw-r-- 1 jsh other 0 Aug 25 15:43 foo
```

Listing 3

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <utime.h>
#define USAGE()
  err_quit("usage: touch [-am] file", argv[0])
#define S_IRW
  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
int mflag, aflag;
main(int argc, char *argv[])
{
        char *cp;
        while (--argc && (**++argv == '-'))
                for (cp = (*argv)+1; *cp != ' '; cp++)
                        if (*cp == 'a') ++aflag;
                        else if (*cp == 'm') ++mflag;
                        else USAGE();
        if (argc < 1)
                USAGE();
        if (!aflag && !mflag)
                ++aflag, ++mflag;
        while (argc--)
                settime(*argv++);
        exit(0);
}
settime(char *filename)
{
        struct utimbuf times;
        struct stat buf;
        if ((access(filename, 0) == -1)
          && (creat(filename, S_IRW) == -1))
                err_sys("creat failed");
        if (stat(filename, &buf) == -1)
                err_sys("stat failed");
        /* now the defaults */
        times.actime = buf.st_atime;
        times.modtime = buf.st_mtime;
        if (aflag)
                times.actime = time(NULL);
        if (mflag)
                times.modtime = time(NULL);
        if (utime(filename, &times) == -1)
                err_sys("utime failed");
}
```

The -c option means "don't create the file if it doesn't exist." How would you implement that?

• The -r filename option to time means "use the time values of this file." How would you implement that?

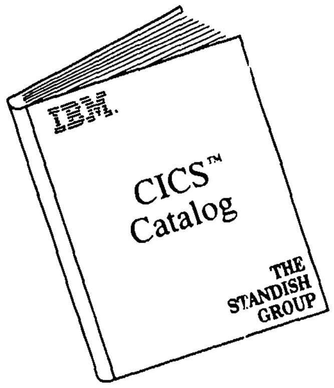
• The -t argument is a time to use for the utime() command. How would you turn that into seconds since the Epoch?

We didn't quite make it to file contents this time, as we promised, but we'll get there next month! We think. ▲

ANNOUNCING
The CICS™ Catalog

1001 ways to get your CICS™

If you use CICS, or plan to, you need this comprehensive catalog. Why? - Because it is the one place to find Who's Who and What's What in CICS. Applications, tools, platforms, system integrators, system programmers and analysts, applications developers and all the latest CICS related software and services.



To get *your* CICS CALL NOW

**1-800-GET-CICS
(1-800-438-2427)**

Available December, 1993 - $50
Major Credit Cards Accepted

If you are a CICS Applications Developer or Service Provider and you're not in the catalog - Call, Write or Fax us now.

The Standish Group International, Inc.
295 White's Path
S. Yarmouth, MA 02664
(508) 760-1800 Fax (508) 760-1830

CICS and IBM are trademarks and registered trademarks of International Business Machines

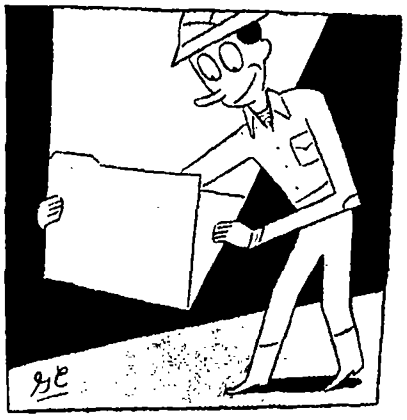
Circle No. 19 on Inquiry Card

In Which We Discuss File Contents (Finally)

by Jeffreys Copeland and Haemer

Welcome back. This is the seventh in our series of columns about the POSIX.1 system interfaces. Instead of approaching this as a list of system calls, we've been working from our knowledge of the POSIX.2–that is, UNIX–commands. By working backwards from the commands, we can derive information about what features the interface to the operating system must provide for the commands to act as they do. The goal is, instead of giving you a guided tour of the standard, to get you to the point where you can tour the standard yourself.

In general, we've been serving columns modeled on sandwiches. We end with a series of points to consider for next time (rye), have a body of current discussion and examples (peanut butter), and begin with solutions to last column's *gedankenexperimenten* (another slice of rye). We're trying to season the mix (pickles) with some observations about good programming practice. In that spirit, let's start with last month's questions.



Revisiting File Attributes

• *If we typedef* `time_t` *to be a signed, 32-bit quantity, when will time end?*

If time begins at midnight on Thursday, 1 January, 1970, 2,147,483,647 seconds later is Tuesday, 19 January, 2038, at 3:14:07 a.m. (Note to NASA: Double-check the declarations in `<time.h>` before

Jeffrey Copeland (`jeff@aus.shl.com` or `copeland@alumni.caltech.edu`) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

starting that three-year flight to Mars in 2035.) Adding another bit, that is, making time an unsigned 32-bit quantity, extends time until Sunday, 7 February, 2106 (happy 121st birthday, Allie) at 6:28:14 a.m.

UNIX is not the only system for which this is a problem: On 4 January, 1975, time ended for users of DEC PDP-10 computers after only 11 years. The solution was to co-opt three unused bits in the status word already containing the date. On 9 January, 1986, more problems surfaced when the second of the three new bits came into use. Lesson: Encapsulate your data structures and don't use undocumented or reserved space.

Listing 1

```
$ grep Jeff *.mm
posix-1.mm:.AU "Jeffreys Copeland & Haemer" "" ""
posix-1.mm:The Jeffs are, we admit, American engineers.)
posix-1.mm:Jeff Copeland (jeff@aus.shl.com) lives in Austin, Texas,
posix-2.mm:.AU "Jeffreys Copeland & Haemer" "" ""
```

• *When should commands take multiple filenames as arguments? no filenames as arguments?*

Let's begin to answer that question with more questions. What commands must process only a single file? When does it make sense to read only from standard input?

Think about csplit for a moment. This command processes a file based on a series of regular expressions and line counts on its command line. It would have confusing semantics if it processed more than one file: Is there a special expression for "somewhere in the next file"? How do I tell when I've begun the next file?

On the other hand, I want to process multiple files in cases where the file boundaries don't make a difference, for example, cat. In some of these cases, I want to mark the file boundaries as in Listing 1.

How about commands that read only from standard input and don't take command-line arguments? These are harder to justify, though they certainly exist (see, e.g., col (1)). We're tempted to say that any command that takes input on the standard input should also accept input from a named file. Can you offer a counterexample? Our email addresses can be found at the end of this article.

• *The POSIX.2 touch command has the following synopsis:*

```
usage: touch [-amc][-r filename|-t [[CC]YY]MMDDhhmm[.SS]] file ...
```

The -c *option means "don't create the file if it doesn't exist." How would you implement that?*

Easy: One possible solution is to exit if a stat() of the file returns -1. Another is to use the access() call we introduced in an earlier column.

• *The* -r filename *option to* time *means "use the time values of this file." How would you implement that?*

Again, stat() provides the operative trick. Instead of setting the time to the current one, use utime() to set the file times to those returned by a stat() of filename.

• *The* -t *argument is a time to use for the* utime() *command. How would you turn that into seconds since the Epoch?*

Slightly trickier. We parse [[CC]YY]MMDDhhmm[.SS] into a tm structure, and then use the Standard C routine mktime(), which takes a tm, and emits a time_t.

But how does mktime() do its work? A long calculation with the count of days since 1/1/1970 is involved, multiplication by 60*60*24, and then addition of hour, minutes and seconds since midnight is called for. The day-count calculation is the hardest. In general, when a date calculation is involved (and between us, we've written too many time sheet and calendar systems), we like to refer to the formulas and common Lisp code in Dershowitz and Reingold's paper "Calendrical Calculations," *Software: Practice and Experience*, 20 (9), September 1990, pp. 899-928. (Aside for the month: Why are Easter and Passover either in the same week or nearly a month apart? If you look at how the dates for these movable feasts are calculated, it becomes clear.)

Lesson: Encapsulate your data structures and don't use undocumented or reserved space.

Finally, File Contents

We've been talking about talking about file contents–that is, details of reading and writing–for three columns. Now, we finally make good on the promise. As we'll see, this will give us an excuse to study some details of POSIX.1 implementation.

Consider the handy cat command:

```
$ ls -l /unix
-rwxr-r-  3 root   other   749172 Aug 28 1991
/unix
$ cat /unix > /dev/null
```

Listing 2

```
#include <stdio.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
 FILE *fp;
 if (argc = = 1)
  docat(stdin);
 else
  while (-argc > 0) {
   if ((fp = fopen(*++argv, "r")) = = NULL)
    err_ret("can't open %s", *argv);
   else {
    docat(fp);
    fclose(fp);
   }
  }
 exit(0);
}

docat(FILE *fp)
{
 int c;

 while ((c = getc(fp)) != EOF)
  putchar(c);
}
```

Listing 3

```
#include <fcntl.h>

main(int argc, char *argv[])
{
 int fd;
 if (argc = = 1)
  docat(0);
 else
  while (-argc > 0) {
   if ((fd = open(*++argv, O_RDONLY)) != -1) {
    docat(fd);
    close(fd);
   }
   else
    err_ret("can't open %s", *argv);
 }
 exit(0); }
docat(int fd)
{
 char buf;
 size_t n;
 while (read(fd, &buf, 1) != 0)
  write(1, &buf, 1);
}
```

Calculating the Effect of Buffer Sizes

buffer	time		
size	real	user	system
1	3:12.85	0:31.64	2:29.83
2	1:38.23	0:14.03	1:16.97
16	0:12.09	0:01.56	0:09.82
128	0:01.54	0:00.14	0:01.33
256	0:00.76	0:00.07	0:00.67
1024	0:00.22	0:00.02	0:00.20
2048	0:00.13	0:00.02	0:00.11
3072	0:00.10	0:00.03	0:00.07
4096	0:00.09	0:00.03	0:00.05
5120	0:00.07	0:00.02	0:00.04
6144	0:00.07	0:00.02	0:00.04
7168	0:00.06	0:00.00	0:00.06
8192	0:00.06	0:00.01	0:00.05
9216	0:00.06	0:00.01	0:00.04
10240	0:00.06	0:00.01	0:00.04

We'll write two quick, cat-like programs and try some experiments (see Listings 2 and 3).

What about their relative speeds? We've already identified a handy 750-KB file:

```
$ /bin/time cat1 /unix > /dev/null
real    4.1
user    0.0
sys     0.9
$ /bin/time cat2 /unix > /dev/null
real    7:44.9
user    22.8
sys     6:14.5
```

Why is the second version so much slower? After all, in classic UNIX, the stdio library is implemented on top of the read and write system calls. Why should the library routine be faster?

The issue is buffers. stdio provides buffered input and output and reads a full buffer into memory at a time. On a real UNIX system, the version that uses the naked read() call has to do physical I/O for each character read. Worse, it has to do a seek() to return to the correct place in the file for each character.

Our colleague, Dave Taenzer, first showed us this with a program like our third version of cat in Listing 4.

We run this by specifying a buffer length and a file, for example:

```
$ cat3 1024 /unix
```

(Note that we've defined a default if a size isn't specified.) This program shows us the dramatic effects of varying the buffer size (see "Calculating the Effect of Buffer Sizes").

For a one-byte buffer, this version of cat is glacial in speed. Notice that we have a curve that drops almost exponentially, and then flattens off as the buffer size gets larger. We ran this on AIX: Notice that the curve flattens off at 4,096 bytes–exactly the value of BUFSIZ in <stdio.h>.

Take-Home Lessons

There are a couple of lessons here:

First, on an implementation where system calls and library routines are distinct–and POSIX.1 leaves that as an implementation detail, referring to any member of the API as an "interface"–system calls are not necessarily faster than library calls. As we've seen in this case, that's because of the buffering of the reads. You're certainly likely to force a context switch while waiting for physical I/O to complete–and as we all discover as we get older, context switches get expensive.

We would be remiss if we didn't also derive some useful progamming lessons from these examples. Note that all three versions of cat are pretty much the same program. We have to do a bit less jiggering of files and flags when we use stdio, which is one of the reasons to use the library in preference to the system calls.

Notice that we've been careful to always close our files. In principle, any we've left open get closed on exit, but there are often limits on how many files we can have open simultaneously–in stdio.h how big is the array of FILE structures?

Another important point is that getchar() returns an int, not a char. See our earlier *RS/Magazine* series on internationalization if you need more convincing on this point.

Notice that real time does not equal user time plus system time. In general, the figure of merit is user plus system time; real time is the elapsed–wall clock–time, which is dependent on external factors, such as system load.

As another implementation detail, /dev/null is not the best way to test I/O performance. On some systems, the kernel recognizes when a write is taking place to /dev/null and ignores the operation.

Listing 4

```
#include <fcntl.h>
#include <ctype.h>
#define DEFBUFSIZ   1024

main(int argc, char **argv)
{
 int fd;
 int bufsize;
 int files = 0;

 bufsize = DEFBUFSIZ; /* set the default size */
 while (-argc > 0) {
  if ((fd = open(*++argv, O_RDONLY)) != -1) {
   docat(fd,bufsize);
   close(fd);
   files++;
  }
  else if( isdigit(**argv) )
   bufsize = atoi(*argv);
  else
   err_ret("can't open %s", *argv);
 }
 /* if we haven't seen files yet,
  process the standard input */
 if( !files )
  docat(0, bufsize);

 exit(0);
}

docat(int fd, int bufsize)
{
 char *buf;
 size_t n;

 if( (buf = malloc(bufsize)) = = NULL )
  err_ret( "can't allocate %d buffer", bufsize );
 while ((n = read(fd, buf, bufsize)) > 0 )
  write(1, buf, n); /* remember that fd 1 is stdout */
 free(buf);
 return( 0 );
}
```

Back of the Envelope Calculations

We can't finish up without leaving you with a few points to ponder for a month:

• How would you implement getchar()?

• How would you implement ungetc()?

• Some implementations of stdio only allow one character to be pushed back by ungetc(). Does yours?

• What's a good maximum pushback? Can you make it infinite?

• Now that we've told you you're better off always using the standard I/O library, when would you want to ignore our advice?

• We've made extensive use of the time command this month. How does it know how much time was taken and how it was spent?

Until next time... ▲

In Which We Discover Processes

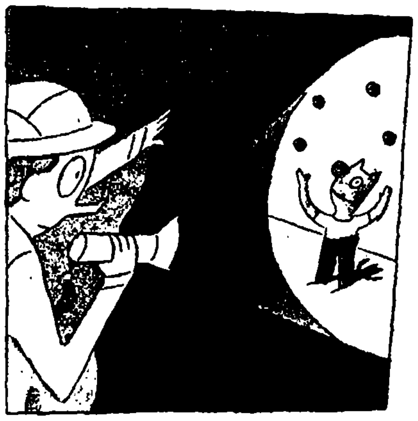
by Jeffreys Copeland and Haemer

Hello again for another in our series about the POSIX.1 system interfaces. This month we'll be reviewing what we know about the file system, before we discuss processes. As those of you who've been here before know, we've been working backwards from our knowledge of UNIX and deriving knowledge about the required system calls to provide the necessary services. In general, we've been packaging information between exercises for the reader, so let's start, as usual, with answers to last month's questions.

File Contents, Redux

• *How would you implement* `getchar()`?

If you were paying attention last month, using the `read()` system call to get one character at a time is not going to work. An obvious buffered solution would set up a static buffer of `BUFSIZ`, read one buffer at a time, and step through that buffer returning characters until the end of that buffer, and then reading another. The code



fragment in Figure 1 illustrates the point.

A better implementation would operate on top of `read()` instead of `fgets()`, so that different standard I/O calls could be more safely intermixed–however, remembering our lessons from last month, an implementation using `read()` would need to always read a full buffer, that is, `BUFSIZ`, at a time.

• *How would you implement* `ungetc()`?

Some implementations of `stdio` allow only one character to be

Jeffrey Copeland (jeff@aus.shl.com or copeland@alumni.caltech.edu) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Figure 1

```
#include <stdio.h>
static char buf[BUFSIZ];
static char *bp;
static int bcount = -1;
int getchar()
{
 if( bcount < 0 ) {
  bcount = fgets( buf, BUFSIZ, stdin );
  if( bcount <= 0 )
   return EOF;
  bp = buf;
 }
 bcount-;
 return *bp++;
}
```

pushed back by ungetc(). Does yours? Most often, in our experience, ungetc() has been implemented as a single-character holding area, local to the module containing getchar(). For example:

```
static int ungetbuf = EOF;
int ungetc( int ch )
{
 ungetbuf = ch;
 return ch
}

int getchar()
{
 if( ungetbuf != EOF )
 {
   int i;
   i = ungetbuf;
   ungetbuf = EOF;
   return i;
 }
 (continue as in the last code fragment)
.
.
}
```

A better implementation would involve a flag, in case we wanted to push back an EOF character. Why not implement this by just pushing back the buffer pointer? Because we may want to push back a character that we hadn't read. Why not allow more than one character of pushback by using the read buffer to push those characters into? After all, we've already read them. Let's answer that with another question: What happens if we want to push back something when we just happen to have read a new buffer? How would we push back before our buffer pointer?

• *What's a good maximum pushback? Can you make it infinite?*

There is an argument to be made for a single-character pushback stack: We always know how big it is, and we have to take care to write our programs to not overflow it. If the pushback buffer is implementation dependent, we can write a program that depends on a different pushback buffer size and quickly get ourselves in trouble. But if we want more than one character of pushback, a buffer full is probably a good compromise. We could make the buffer size infinite, or nearly so, by pushing the characters back into a FIFO, or pipe, or even by opening a file and storing them on disk.

• *Now that we've told you you're better off always using the standard I/O library, when would you want to ignore our advice?*

When you need to use unbuffered I/O. When is that? Well, for openers, if you're implementing a standard I/O replacement. (One of us made the mistake of building a stdio replacement on top of stdio once: The performance was abysmal.) If you need to deal with a character special device, such as a raw communications line, you have to deal in system calls, instead of buffers. The most important case, though, is when you're building a kernel. An important side point: Because stdio calls exist as part of POSIX, the functionality of stdio exists on a lot of non-UNIX systems. For example, most C compilers on DOS have stdio, without the benefit of a UNIX kernel.

• *We've made extensive use of the* time *command this month. How does it know how much time was taken and how it was spent?*

You might be surprised to know that there's a POSIX.1 interface that returns the system and user time expended by a process. Then again, if you've been paying attention, you're probably not surprised at all there is to learn about the times() call. It returns a structure containing the number of clock ticks used so far by this process for user and system time, and the number of clock ticks used by the children of this process that have terminated. Clock ticks are defined in terms of seconds in <time.h>. From there, and with the discussion of process creation in this column, we leave it as an exercise for the reader to implement the time command.

The File System

Let's quickly review the file system: The vast bulk of what we rely on the operating system for is dealing with files. We read them; we write them; we find out when they were last modified. We want the operating system to control access to files for us.

We've talked about interfaces to tell us about files, such as stat() and access(). We have interfaces to help us reset file information, such as chmod() and utime(). We've used the set of file creation routines: open() and creat(). We can also destroy files with remove() and unlink(). We know how to handle special files, such as directories, with mkdir(), opendir() and friends.

Figure 2

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>

main(void)
{
    pid_t pn;
    char *type;

    printf("initially: pid = %d, ppid = %d\n",
        getpid(), getppid());
    if ((pn = fork()) == -1)
        err_sys("can't fork");
    type = pn ? "parent" : "child";
    printf("%s (fork returns %d):"
        "pid = %d, ppid = %d\n",
            type, pn, getpid(), getppid());
    exit(0);
}
```

Let's look at the output:

```
$ a.out
initially: pid = 26411, ppid = 26410
child (fork returns 0): pid = 26412, ppid = 26411
parent (fork returns 26412): pid = 26411, ppid = 26410
```

Now that we know what file system things we have in POSIX.1, what are we missing in the specification?

POSIX doesn't talk about specific files. For example, there is nothing in the spec about the format of core files. We don't need to have a /tmp directory, or /dev/null device. As we've discussed before, we don't even need to store our user information in the traditional /etc/passwd; all we need to be able to do is retrieve that user information with getpwnam().

POSIX doesn't lay out uids or groups. For example, we don't need to have a user id 0 for root, or the daemon group.

POSIX doesn't tell us how the kernel has to work. As we've pointed out before, we make no assumptions in POSIX.1 about which interfaces are system calls, and which are library routines. We don't have anything in the spec about the data structures in the kernel, for example, disk layout, or inodes, or superblocks.

POSIX.1 doesn't talk about symbolic links. They were not sufficiently widespread standard practice when the POSIX effort started.

POSIX.1 doesn't talk about mount points or the mount() system call. Those are left as implementation dependencies.

File Systems + Processes = Operating Systems

With a tip of the hat to Niklaus Wirth, we begin to discuss processes. Consider the program in Figure 2.

What's a fork()? Are there spoon() and knife() interfaces, too? (No.)

The fork() creates a process: It duplicates the process space, and both the old program (the parent) and the new program (the child) continue running from immediately after the fork() call. The new program retains all the file pointers, and environment of the original.

How does the program know whether it's the parent or the child? fork() returns 0 to the child, and the process id of the new process to the parent. So when our program executes, the first printf() executes once, and the second printf() prints once for each process.

(What if the fork fails? It returns -1. Why might the call fail? If there's a limit on the number of processes the system can have running simultaneously. If the new process is going to take too much memory–for example, it would cause us to run out of swap space.)

Let's try a more complicated example, using two views (see Figure 3). Assume that the ps commands are running on another terminal, and we'll take a look at the processes running on the console as we try to log in.

How come the processes all have the same process id, even though they're running different programs? Because we don't do a fork().

Figure 3

Console Login:

```
$ ps -f -t console
 UID   PID    PPID  C  STIME     TTY      TIME  COMMAND
 root  19953  1     0  16:30:03  console  0:00  /etc/getty console console
```

Console Login: jsh
Password:

```
$ ps -f -t console
 UID   PID    PPID  C  STIME     TTY      TIME  COMMAND
 root  19953  1     0  16:30:03  console  0:00  login jsh
```

Console Login: jsh
Password:
... / successful login */*

```
$ ps -f -t console
 UID   PID    PPID  C  STIME     TTY      TIME  COMMAND
 jsh   19953  1     2  16:30:03  console  0:03  -bash
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/limits.h>
main(void)
{
   char line[_POSIX_MAX_CANON];

   printf("Console Login: ");
   fgets(line, _POSIX_MAX_CANON, stdin);

   execl("./login", "login", line, NULL);
   err_sys("exec failed");
   exit(EXIT_FAILURE);
}
```

Notice that the program reading the user name at the login prompt is different from the program that reads the password, and that program (login) actually begins a separate program: the shell. These hints should give us some vague idea of what execl() does. It replaces the current image with a new executable. There are about 57 varieties–well, actually only six–of exec(): execl, execv, execle, execlp, execve, execvp. Versions ending in l take a NULL-terminated argument list; the versions ending in v take an array of arguments; p means to use the search path for the executable; e versions provide an argument for the environment array. In all versions, the second argument is the value the process receives as argv[0]. OK, so now that we know that the execl(...) in the above example runs a new program, let's look at that program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/limits.h>
#include <pwd.h>
#include <string.h>
static struct passwd *pwd;

main(int argc, char *argv[])
{
 int i;
 char logname[_POSIX_MAX_CANON];
 char *getpass(), *password;

 strcpy(logname, argv[1]);
 for(i=0; ; i++) { /* infinite loop */
  *(logname + strlen(logname) - 1) = ' ';
  pwd = getpwnam(logname);
  password = getpass("Password: ");
  login(logname, password); /* won't return */
  if (i > 4)
   break;
  printf("login: ");
  fgets(logname, _POSIX_MAX_CANON, stdin);
 }
 sleep(20);
 err_quit("Too many login failures");
 exit(0);
}
```

Notice that the sleep() interface just suspends operation for some number of seconds. Also, notice that we've used fgets() to read the logname, not gets(). The former routine is safer: We just need to remember that fgets() returns a string containing the line-terminating newline. (Remember that the Internet worm half a dozen years ago was transmitted by causing a gets() to read into memory past the end of the buffer allocated–fgets() never reads more than the given number of characters.) What are we going to do if the login attempt fails? We'll let the new program handle that.

```
login(char *logname, char *password)
{
 char shellname[128];
 char *basename();
 if (passmatch(logname, password)) {
   if (chdir(pwd->pw_dir) = = -1)
     err_sys("chdir");
   strcpy(shellname, "-");
   strcat(shellname, basename(pwd->pw_shell));
   execl(pwd->pw_shell, shellname, NULL);
   err_sys("exec failed");
 }
 sleep(5);
 err_msg("Login incorrect");
}
```

Note that chdir() changes the working directory. Also, we invoke the shell with a "-" prepended: This is how a shell tells it's a login shell, when it examines its argv[0].

```
passmatch(char *logname, char *passwd)
{
 /*
 encrypt entered password,
 get encrypted password corresponding to login,
 compare the two
 if they're equal, return(1)
 */
 return(1);
}
```

passmatch() is a little sketchy because POSIX.1 doesn't deal with security-related issues like password encryption. This is a special case of the general POSIX approach of not dealing with anything hard or controversial.

```
char *getpass(char *prompt)
{
    static char password[_POSIX_MAX_CANON];

    printf("Password: ");
    fgets(password, _POSIX_MAX_CANON, stdin);
    return(password);
}
```

This echoes the password on the screen. Can we do better?

```
#include <termios.h>

char *getpass(char *prompt)
{
    static char password[_POSIX_MAX_CANON];

    struct termios oterm, nterm;

    tcgetattr(0, &oterm);
    nterm = oterm;
    nterm.c_lflag &= ~ECHO;
    tcsetattr(0, TCSAFLUSH, &nterm);
    printf("Password: ");
    fgets(password, _POSIX_MAX_CANON, stdin);
    tcsetattr(0, TCSAFLUSH, &oterm);
    putchar(' );
    return(password);
}
```

Points to note:

- `tcgetattr()` and `tcsetattr()` are new to POSIX. They were invented because `ioctl()` was too big a grab bag, including everything but control bits for the kitchen sink.
- General style of use for these new routines is to store away the old settings, modify a copy of the settings, do all the necessary work, restore the original settings.
- `stty` is built on these two calls.

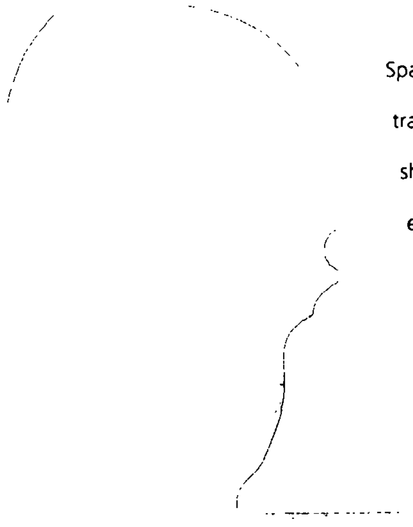
Questions to Leave You With

Since we've barely had time to begin discussing processes, and `fork()` and `exec()`, and have thrown quite a bit of code at you, we leave you with only two questions:

- What happens if we interrupt when echoing is turned off? Have you ever done anything like this? How do you fix it?
- UNIX (but not POSIX) supplies a `getpass()` function. What does it do that we don't?

That's all, folks. We'll continue with processes next month. ▲

EL CAMINO REAL



Spanish for "The Royal Highway", it was followed by **DEVOTED** Franciscan Monks traveling between the California missions, and was no more than a dusty path marked by the shepherd's staff and bell. In the English language, Websters defines **REAL** as "actually existing, not fictitious or imaginary, **GENUINE**, true, fixed, immovable." As one of **IBM**'s top revenue producing industry remarketers for 1992 and 1993, **REAL** fits this definition. We've **DEVOTED** our business to providing our **CUSTOMERS** with the best hardware, software and technical support available. Whether you are a customer or an **AFFILIATE**, you'll get royal treatment from **REAL**. Call today for **MORE** information on our Distribution & Financial Accounting products, our **LOTS** services (Level One Technical Support), as well as non-proprietary products we sell and **SUPPORT** on the AS/400 & RS/6000.

(800) 359-3475

REAL APPLICATIONS, LTD.

an El Camino company

IBM is a registerd trademark of the International Business Machines Corporation.

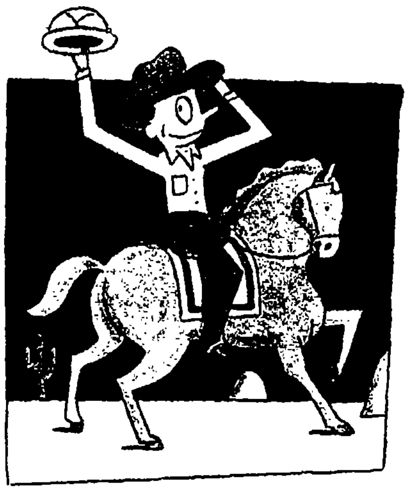
Circle No. 14 on Inquiry Card

In Which We Corral Some Processes

by Jeffreys Copeland and Haemer

Well howdy, buckaroos. See? Out West we really do talk like that. (In the interests of candor, we really only talk like that when we're played by Roy Rogers and Gene Autry and are writing columns or giving presentations to customers or managers. When we program, we mumble "#$$@#%#% memory leaks" a lot and are played by Clint Eastwood and Lee Van Cleef.)

We're back to tour some more POSIX.1 system interfaces. Last month, we moseyed on over to processes from files. We'll look at processes some more. But first, a word from our sponsor–last month's column.



Hello? Hello?

Last month, we showed a simple-minded implementation of the login process: `getty` execs `login`, which validates the password and then execs a shell. While we were getting the user's password, we used `tcsetattr()` to turn off echoing, then turned echoing back on again after prompting for a password, and storing away the answer. In UNIX systems, this entire job is performed by the `getpass(3)` function, which returns the prompted-for password. The sample we wrote differs from the one typically supplied in a few trivial ways and one important way.

The trivial ways? Our `getpass()` sent its password prompt to `stdout` and reads a response from `stdin`; the manual page on our system says the prompt should go to `stderr` and the response read from `/dev/tty`. Also

Jeffrey Copeland (`jeff@aus.shl.com` or `copeland@alumni.caltech.edu`) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

from last month, our password buffer is declared to be of size _POSIX_MAX_CANON. The man page says passwords can be of length _PASSWORD_LEN. Not big changes, but you get the idea. (We actually left in these differences deliberately, in hopes that some of you would contrast the code with the man pages and try out some changes.)

The important way? Signal handling. If you've ever bailed out of vi, or some other raw-mode application, in an awkward way and left your terminal in a state where typed characters won't echo, you'll know what we mean. At first, when you get in this state, you try to type things, get no response, and panic. Later, you discover that ^J

Listing 1. Finding Processes

```
$ ps -axj |
awk ' {
  pid=$2; ppid=$1
  $1=pid; $2=" " ppid
  for (i=3; i<10; i++)
     $i=""
  print $0
}'
```

Output:

```
PID     PPID    Command
0       0       swapper
1       0       /sbin/init -
2       0       pagedaemon
42      1       portmap
57      1       (biod)
58      1       (biod)
59      1       (biod)
60      1       (biod)
70      1       /usr/lib/sendmail -bd -q30m
76      1       rpc.mountd -n
77      1       (nfsd)
80      77      (nfsd)
81      77      (nfsd)
82      77      (nfsd)
83      1       rpc.lockd
85      1       rpc.statd
88      1       ./erpcd
91      1       ./snmpd
94      1       ./ntpd
97      1       update
100     1       cron
102     1       inetd
105     1       /usr/lib/lpd
2970    1       named
4860    1       SCREEN
5443    102     in.rlogind
18861   1       /usr/etc/syslogd
19983   102     in.rlogind
24939   1       -sendmail bell (sendmail)
24940   24939   mail -d bell
25511   102     in.rlogind
25980   1       /usr/local/bin/elm.real
26074   25980   sh -c /usr/local/bin/med /tm
26075   26074   /usr/loc
27532   102     in.rlogind
28465   102     in.rlogind
28932   102     in.rlogind
28978   102     in.rlogind
29093   102     in.rlogind
29193   102     in.rlogind
29452   102     in.rlogind
29765   102     in.rlogind
12156   1       - std.9600 console (getty)
27533   27532   -tcsh (tcsh)
9765    25512   rlogin csn
9766    9765    rlogin csn
25512   25511   -csh (csh)
28466   28465   -csh (csh)
28509   28466   layers
29766   29765   -csh (csh)
29776   29766   /usr/ucb/mail
29936   29776   vi /tmp/Re29776
29094   29093   -bash (bash)
29958   29094   ps -axj
19984   19983   -tcsh (tcsh)
29453   29452   -csh (csh)
29462   29453   /bin/csh /usr/local/bin/menu
29466   29462   /usr/local/bin/elm.real
29576   29466   sh -c /usr/local/bin/med /tm
29577   29576   /usr/local/bin/perl /usr/loc
28933   28932   -csh (csh)
29195   29193   -csh (csh)
29937   29195   /usr/ucb/mail haviland eric
5444    5443    -csh (csh)
28521   28509   -bin/csh (csh)
29634   28521   /usr/ucb/mail
29949   29634   <defunct>
29950   29634   <defunct>
28527   28509   -bin/csh (csh)
28569   28527   vi infscript
28531   28509   -bin/csh (csh)
28542   28531   vi changes.920827
4864    4860    -bin/csh (csh)
28979   28978   -csh (csh)
28995   28979   rn -m
```

works where Enter used to and that `stty sane ^J` makes your terminal sane enough to let you get your affairs in order, log out and log back in again. What's happened is that your raw-mode process exited–almost invariably because of a signal–before restoring the terminal to cooked mode with `tcsetattr()`. In cooked mode, the terminal driver echoes the characters you type, and translates between the carriage returns you type on the keyboard (`^M`, or `\r` in C parlance), the newlines that you specify in C programs (`^J`, or `\n`) and the newline-carriage return pair that you see the cursor perform. In raw mode, the application handles all that work. Once you've set the terminal into raw mode, most applications that expect to be running in cooked mode and are waiting for a `\n` won't hear one unless you type a `^J`. To fix this, you need to run `stty`, which just parses and interprets its arguments and then calls `tcsetattr()` and friends. Naturally, you need to end the command with `^J`, since `stty` won't execute until after the shell sees your command, which it won't until you terminate the line with a `\n`.

Real raw-mode applications protect themselves against this problem by having signal handlers that restore the terminal modes before exiting. Our code doesn't. We'll talk about how to handle signals later in the series.

You say you've never had this problem? You haven't been experimenting with your system enough.

Processes

OK, so we can use the `exec()` family to make a chain of new processes. So what? Even DOS–heck, even CP/M–could do that much. Something more is at work: a `fork()` followed by an `exec()`, which first clones a process, then overlays one of those processes–typically, the child–with a second program. This two-step birth of new program from an old one is how UNIX creates its process tree. How useful is this? Is creating a new process rare and arcane, or commonplace? Let's look at a real, running system: `teal.csn.org`, one of Colorado SuperNet's Sun systems. First, we'll use a shell script to get a list of all processes (see Listing 1).

We note, in an aside, that the script we used to create this listing illustrates why standards are useful. `teal.csn.org` is a BSD-based system. If your machine uses System V, you'll have to use a completely different set of ps flags to get the data. Unfortunately, POSIX has not yet standardized the ps command. In contrast, awk is standardized so that part of the script is portable, as long as your ps output is identical. Unfortunately, on a particular System V-based UNIX (we'll use AIX for our example) you'll need to use something like:

```
ps -fade | cut -c9-20,47- | \
   awk ' { print $1 " " $2 " " $3 }' | sed -n 1,10p
```

(You need the `cut` command because both `ps -l` and `ps -f` on System V produce a variable number of fields in the awk sense. For example, a start-time column is produced in either the form `18:20:23`, or if the process was created more than 24 hours ago, `Feb 07` (see Listing 2).

Listing 2. Synchronizing Start Times

```
USER      PID    PPID    C      STIME      TTY    TIME   CMD
root        1       0    0     Feb 04        -    5:14   /etc/init
ricks    1970       1    0   10:10:34    hft/0    0:01   -ksh
root     2560       1    0     Feb 04        -    1:09   /etc/syncd 60
root     2993    6319    0     Feb 04        -    0:00   /usr/etc/biod 6
jeff     3272   35013    0   08:09:39   pts/13    0:01   rlogin alumni.caltech.edu
root     3330       1    0     Feb 04        -    0:00   /usr/lib/errdemon
root     4478       1    0     Feb 04        -    0:00   /etc/srcmstr
root     4980    4478    0   15:45:17        -    5:09   /etc/qdaemon
root     5263    4478    0     Feb 04        -    0:00   /usr/lpd/lpd
root     5587       1    0     Feb 04        -    0:00   /etc/uprintfd
root     5778    4478    0     Feb 04        -    0:00   /usr/lib/sendmail -bd -q30m
root     6028    4478    0     Feb 04        -    0:01   /etc/syslogd
root     6319    4478    0     Feb 04        -    0:00   /usr/etc/biod 6
root     6552    4478    0     Feb 04        -    0:00   /usr/etc/portmap
root     6811    4478    0     Feb 04        -    0:02   /etc/inetd
root     7117       1    0     Feb 04        -    1:23   /etc/cron
root     7378    4478    0     Feb 04        -    0:00   /etc/writesrv
root     7584    4478    0     Feb 04        -    0:15   /etc/rwhod
root     7843    4478    0     Feb 04        -    0:10   /usr/sbin/snmpd
```

Listing 3. Only awk Can Make a Tree

```
0
     1
          42
          57
          58
          59
          60
          70
          76
          77
               80
               81
               82
          83
          85
          88
          91
          94
          97
          100
          102
               5443
                    5444
               19983
                    19984
               25511
                    25512
                         9765
                              9766
               27532
                    27533
               28465
                    28466
                         28509
                              28521
                                   29634
                                        29949
                                        29950
                              28527
                                   28569
                              28531
                                   28542
               28932
                    28933
               28978
                    28979
                         28995
               29093
                    29094
                         29958
               29193
                    29195
                         29937
               29452
                    29453
                         29462
                              29466
                                   29576
                                        29577
               29765
                    29766
                         29776
                              29936
          105
          2970
          4860
               4864
          18861
          24939
               24940
          25980
               26074
                    26075
          12156
     2
```

Similarly, the process wait information is missing from the ps -l output if the process is actually running. Fortunately, the column positions are fixed, even if the field count isn't. We still use awk to make the output conform to the Sun example, but strictly speaking we don't need it. We'll use a second awk program to reprint the data as a tree (see Listing 3).

Here's that awk program, in case you want to use it on your own system.

```
co #!/usr/bin/awk -f
{
  lookup($1)
  lookup($2)
  if ($1 = = $2)
    next          # can't be your own parent
  if (child[$2] = = -1) {
    child[$2] = $1
    next
  }
  for (nextchild = child[$2]; \
    sib[nextchild] != -1; \
    nextchild = sib[nextchild])
    ;
  sib[nextchild] = $1
}

END {
  tprint(0, 0)
}

function tprint(root,indent, i, n) {
  for (i=0; i<indent; i++)
    printf("%5s", " "
  printf("%5s , root)
  for(n = child[root]; n != -1; n =sib[n])
    tprint(n, indent+1)
}

function lookup(node) {
  if (!seen[node]) {
    seen[node] = 1
    sib[node] = -1
    child[node] = -1
  }
}
```

Both of these programs require "new awk," the awk described in *The Awk Programming Language* and standardized by POSIX (Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger, *The Awk Programming Language*, 1988, Addison-Wesley Publishing Co., ISBN 0-201-07981-X). In most of this series, when we say POSIX, we mean POSIX.1, the system interfaces. Commands, however, are standardized in IEEE 1003.2-1993 (POSIX.2).

ANSI-C and POSIX.1 are tools to help you write portable C programs. POSIX.2, in contrast, helps you write portable shell scripts. Although the premise of this series is that knowing the shell-level commands will help you understand the system interfaces, the two standards do not depend on each other. You can have a complete POSIX.2 implementation without an underlying POSIX.1. For example, Mortice-Kern Systems of Waterloo, Ontario, sells a complete dot 2 tool kit, including a Korn shell, make and awk, to run on either DOS or OS/2, neither of which conforms to POSIX.1. On such systems, your C programs may not work, but all your scripts are portable, and at the shell level, the systems feel just like UNIX. It's just such software that allows us to use DOS on our laptops without tearing our hair out–or sounding like Clint Eastwood and Lee Van Cleef.

So what can we see in this tree?

First, process 0 (the swapper) is the root of the entire tree. All other processes could be created by `fork()`, but someone had to create this process by hand. ("A goose egg" is American slang for a zero, suggesting the solution to the age-old riddle about which came first...) Process 1 (`init`) comes next. Notice that almost everyone else is a child of `init`, but that `init` is forked from swapper, not just exec'd. If that weren't the case, the swapper wouldn't show up in the process tree, and it would stop running as soon as `init` started. Second, processes really do form a nontrivial tree. From this, you can infer that the important questions for processes will be the same as those for file systems:

- How do we create and delete nodes?
- What do nodes contain and how do we manipulate their contents?
- What are the other properties of nodes and how do we get and set them?

Until We Meet Again

Over the next few months, we'll explore these questions in some depth. Meanwhile, when you have a spare moment, after you've hog-tied an ornery critter with some old nine-track magnetic tape, or later, when you're sitting around the campfire brewing up a fresh pot of Jolt, think about this:

- Who creates process 0?
- Why are most of the process numbers above so big?
- What's a process that's <defunct>?
- How small a typeface can the average person read?

And, until we meet again, Happy Trails. ▲

WE BUY, SELL & RENT

RS/6000

PARTS • FEATURES
UPGRADES • CPU'S

- System 36 Conversions
- AutoCad Available for RS/6000
- Sun Sparc Compatibles

New & Used IN STOCK
Complete Technical Center, Installation,
Stock Parts & Features for RISC.

COMPUTER MARKETPLACE

(800) 858-1144

A Publicly Traded Company NASDAQ: MKPL

***MOTOROLA UDS** Western Regional Distributor of the Year*

CDLA IBM® as cd

1490 RAILROAD ST., CORONA CA 91720
TEL (909) 735-2102 • FAX (909) 735-5717

Circle No. 7 on Inquiry Card

RS/6000 users...

Overnight Shipping!!

Cambex

is the only choice.

- HD1 & HD3 memory 32 - 128MB
- SCSI disk subsystems
- AIX software utilities
- Tape back-up
- RAID disk subsystems

Cambex is the **ONLY** alternative manufacturer of HD1 and HD3 compatible memory boards for your RS/6000 system.

CALL TODAY for a quick price quote from our Cambex***Direct*** representatives.

1-800-292-RISC

VISA MasterCard

Circle No. 6 on Inquiry Card

In Which We Look at the Processes in the Corral

by Jeffreys Copeland and Haemer

Howdy again, buckaroos. Last time around the campfire, we talked about process trees. As usual, over our last cup of bad campfire coffee, we left you with some questions:

• *Who creates process 0?*

The egg. No, wait. The chicken. No, wait ... It's true. Processes usually only come from other processes, but someone has to make the first process. On typical UNIX systems, process 0–the swapper–is handcrafted by the kernel, as is process 1, `init`. Everyone else is a descendant, direct or indirect, of `init`. (Oh, and the answer is "the egg.")

• *Why [were] most of the process numbers [in last month's column] so big?*

Process numbers are allocated serially. Even though most processes eventually terminate, the process numbers for new processes continue to rise until they hit the maximum value of a `pid_t`–typically either a `short` (64K) or an `int` (4G).

• *What's a process that's <defunct>?*

Imagine you're a process that's forked a subprocess. After that subprocess terminates, you'll want to collect information from it on a variety of things. As a programmer, you've seen and thought about this one before. For example, you've called `exit(8)` in any number of programs, from "hello, world" on. It's the parent process that's interested in the exit status returned by `exit()`. Typically, parents collect

Jeffrey Copeland (`jeff@aus.shl.com` or `copeland@alumni.caltech.edu`) *lives in Austin, TX, where he manages projects for SHL Systemhouse. He recently acted as software consultant for the administrators of the 1993 Hugo Awards. His technical interests include internationalization and typesetting.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

these pieces of information from their children with the system call wait(). But suppose the child terminates before the parent gets to that system call. (Remember uniprocessor UNIX is a time-sharing system. All those processes may seem like they're executing at once, but face it: At any given instant, only one process has the CPU.) Even though the child has terminated, the kernel has to keep around a process data structure for the child until the parent executes the wait() to collect the data it contains. Such ephemeral unprocesses are called "zombies." The data structure in which the data are kept is the same data structure that ps looks at, so ps reports their existence, but marks them "<defunct>." We think it's a pity that ps doesn't just mark them "<zombie>."

One of us (JSH) knows all the verses to "Zombie Jamboree." Oh, and if you haven't seen "Incredible Strange Creatures Who Stopped Living and Became Mixed-Up Zombies," starring Cash Flagg and Carolyn Brandt, you spent too much of your college career studying.

• *How small a typeface can the average person read?*

About five-point type without eyestrain. It's easier with a serifed type, like the one this column is set in. Also, a typeface with a large relative x-height, such as Stanley Morison's brilliant New Times Roman, can be read in a smaller size than an older classic typeface like Goudy Old Style. (What's an "x-height"? That's the height of letters without ascenders or descenders, like x, e and w. New Times Roman has an x-height that's a larger proportion of the total height of the font than most older typefaces.)

Information about Processes

We've said this before, but it's still true: processes + files = operating systems. (There's a book title in there, if no one from ETH Zurich has thought of it yet.)

We spent a whole lot of columns talking about files, and how to get information about them. We talked a little bit two months ago about how to create processes (and we'll come back to process creation in gory detail later). We discussed process trees last month. This month, we discuss the process equivalent of ls: ps.

```
$ ps
   PID       TTY      TIME     CMD
  14418     pts/7     0:00     vi posix-a.mm
  16469     pts/7     0:00     ps
  32025     pts/7     0:00     -ksh
$ ls -l /bin/ps
-r-xr-sr-x     1 bin      bin       46073 Apr 17 1993 /bin/ps
```

Why is ps set-uid? Because it needs to look at kernel data, in /dev/kmem, which is protected from casual perusal. If we could read the kernel data, we could conceivably read data from other users' programs that they didn't want us to read. But why do we need to read raw kernel data? Because there's no process equivalent of the ever-useful file interface, stat().

However, by reading /dev/kmem, ps does allow us to get process information for processes other than our own, so we can get the list of all the processes on the system. If we didn't have access to this information, we couldn't have produced the process tree diagram for the whole system last month.

Be warned, however, that ps can find out all the useful information about your process. For example, it knows about and prints your command-line arguments. A shell script that gets a file from another host by taking the other host's password as an argument would be a security hole.

Unlike every other program we've discussed, ps is not standardized–it doesn't appear in POSIX.2. Worse, it's not just a matter of slightly different flags on the BSD and System V versions; the flags are almost completely different. For example, on BSD, we typically type ps -ajx, but on AIX or other System Vs, we type ps -fade. See Table 1 for further comparison.

As you can see from the table, the commonality between the two versions is almost nonexistent. It's also obvious once again that AIX came from System V roots and has been modified slightly.

Available Information

Skipping over details about what flags we use to get them from ps, what can we find out about processes on the system? Lots, it turns out.

We can get the executable code, and the arguments and the environment, which we get by reading them from memory. The process tree information is available, as we demonstrated last month–that is, the current process' process-id-number, id of its parent, process group. We can find out what signals the process is waiting for, what its signal mask is, how much time is left on any alarm signals in its queue–signals are a topic we'll discuss later in more detail. We also have access to the sort of accounting information for the process that would allow us to bill time–if you can imagine a UNIX system actually billing for time–such as user name, process times and elapsed time. (We know that there are UNIX installations that bill for time, such as commercial Internet providers. It's just that as UNIX hackers from way back, the concept will always be a little foreign to us.) Also, we can get the working directory of the process, the login directory of the process and the root directory of the process group. We aren't going to write ps, but we can experiment with some of the ideas we would need to implement it. Consider the following program:

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>
main(int argc, char *argv[])
{
  struct tms tbuf;
  clock_t ticks;
  char *basename();
  ticks = times(&tbuf);
  sleep(10);
  printf("%6s %-8s %4s %s ,
    "PID", "TTY", "TIME", "COMMAND");
  ticks = times(&tbuf) - ticks;
  printf("%6d %-8s 0:%02d %s ,
    getpid(), basename(ctermid(NULL)),
    ticks/CLK_TCK, argv[0]);
  printf("%6d %-8s 0:%02d %s ,
    getpid(), basename(ttyname(0)),
    ticks/CLK_TCK, argv[0]);
  fflush(stdout);
  system("ps");
  exit(0);
}
```

Some points to note about this program:

- `getpid()` is a process primitive: It returns the process id number.
- `sleep()` waits for the specified number of seconds.
- `times()` returns the time in clock ticks from an arbitrary starting point. (Note the difference from `time()`, which returns seconds from a universally known time.) `times()` also fills a `tms` buffer with process times, which allows the `time` command to work.
- Clock ticks are arbitrary. The manifest constant `CLK_TCK` in `times.h` provides the number of clock ticks per second. (Actually, use of `CLK_TCK` is obsolete–or obsolescent, as we say in standard-speak. The POSIX.1 standard specifies that the value should be the same as the result of a `sysconf(_SC_CLK_TCK)` call.)
- We are trying both interfaces that return the terminal name: `ctermid()`, which gets the pathname to the "controlling terminal," and `ttyname()`, which returns the pathname to the terminal associated with a given file descriptor.
- Last, the `system()` call is a POSIX.2 interface that executes a command line.

What happens when we run this program?

```
$ a.out
   PID    TTY       TIME    COMMAND
 48666    tty       0:10    a.out
 48666    7         0:10    a.out
```

Table 1. A Flag Isn't Always a Standard

Flag	System V	AIX	Sun/BSD
-a	all processes, except process group headers and processes not connected to a terminal	all processes, except process group headers and processes not connected to a terminal	include processes not owned by you
-A	n/a	all processes	n/a
-e	every process	every process (except kernel processes)	display the environment in addition to the command arguments
-f	full listing	full listing	n/a
-F	n/a	custom format	n/a
-g	n/a	n/a	"interesting processes"–normally, the getty waiting for login on an idle terminal, and login shells are not included in the listing
-l	long listing	long listing	long listing
-r	n/a	n/a	"running processes"--runnable processes, processes in page wait (that is, waiting for memory) and processes in short-term noninterruptible waits (e.g., sleep())
-x	n/a	n/a	include processes without a controlling terminal

```
  PID    TTY     TIME    CMD
18669    pts/7   0:00    -ksh
19228    pts/7   0:00    ps
48666    pts/7   0:00    a.out
50459    pts/7   0:00    bsh bsh bsh
```

Can we identify the processes? The -ksh is the login shell under which we are running. The ps is the program executed by the system() call; and the bsh is the shell created to run that command. a.out is our test program.

Notice that ctermid() returns /dev/tty, which is not the most helpful information in the context. We'd like to know what device we're actually using. In fact, since we're actually running an X terminal on a pseudo-tty, that column should at least say pty rather than tty. At least ttyname(0) provides us with the pseudo-tty number, even if it doesn't provide us with as complete information as the real ps.

But, even worse, if we provide our program with a standard input, for example, by running it from within the editor, we get:

```
  PID    TTY     TIME    COMMAND
20790    tty     0:10    a.out
20790            0:10    a.out
  PID    TTY     TIME    CMD
18669    pts/7   0:00    -ksh
20790    pts/7   0:00    a.out
38945    pts/7   0:00    vi posix-a.mm
50488    pts/7   0:00    ps
50743    pts/7   0:00    bsh bsh bsh
```

(Note the addition of a vi session to the process listing this time.)

Notice that there is no tty name on the second line. When standard input is attached to a file, as it is when we run a command like !!a.out from inside vi using a.out as a filter on the line we're pointing to, there is no controlling tty for file stream zero, or stdin.

So how does ps manage to get the correct tty name? Magic. Once again, the ability to probe into kernel data gives ps powers beyond those of mortal men, but here's an interesting case in which there is no standard interface that even lets a process ask about its own value of something that the kernel has access to. Most systems have programs that can provide even more detailed information: On AIX, most of the smit service programs provide good examples; on other systems, look for programs with names like kdb, kdump, fsdb or crash.

Points to Ponder

In parting, we'll leave you with our usual question or two:

- If process ids are assigned sequentially, why is the id of the shell, in our example above, higher than the id of the ps that it's running?
- Why isn't there a process-level equivalent of stat()?
- Are all the characters in your name legal flags to ps? (On at least one system we're running, "ps -haemer" works fine.) If so, what kind of output do you get?
- Why would ctermid() return /dev/tty instead of something useful? ▲

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
Features:	High	Medium	Low
Big Business	170	171	172
PC-to-UNIX Mail Integration	173	174	175
Security Is...	176	177	178
Looking Under the Hood	179	180	181
Columns:			
Q&AIX–Be Careful Out There	182	183	184
Systems Wrangler–X and the Sysadm	185	186	187
Datagrams–Sudo	188	189	190
AIXtensions–ODMspeak	191	192	193
POSIX–A Look at the Processes in the Corral	194	195	196

In Which We Go to the Beach

by Jeffreys Copeland and Haemer

Now is the time to take the blanket, pack the picnic basket and head for the beach. Unfortunately, from where we live now, it's a multiday trek. Nonetheless, we can do a little exploring from where we sit. Actually, we're not so much interested in the beach as the shells on it: We'll be exploring the POSIX `exec()` interface and how it's used in the shell. We'll write a simple version of the shell to demonstrate. (Yes, it's a hackneyed pun, but it gave us a headline this month, didn't it?)

A Quick Visit to the Process Corral

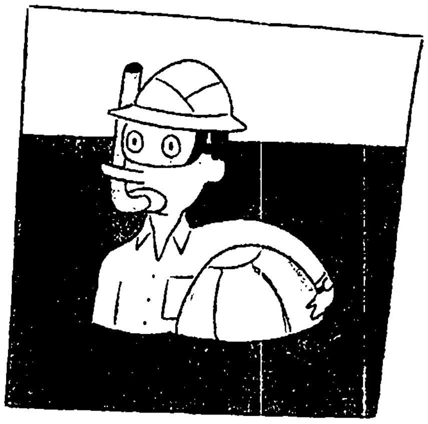
First, as usual, we'll review the questions we left you with at the end of last month's column:

• *If process ids are assigned sequentially, why is the id of the shell in our example higher than the id of the* `ps` *that it's running?*

Yes, process ids are assigned sequentially, but after you've assigned the 65,535th process, we cycle back to zero. No, actually, that's the swapper, so we cycle back to process 1. No, actually, that's still `init`, which creates processes, so we go from pid 65535 to pid 2.

• *Why isn't there a process-level equivalent of* `stat()`*?*

There easily could be. However, POSIX is not intended to create a better operating system, but rather to codify existing practices because there's no `pstat()` in existing systems, there's none in POSIX.1. A more interesting question is why didn't Ken Thompson include one in the original UNIX implementation? For that question, we will have to plead ignorance.



• *Are all the characters in your name legal flags to* `ps`*? (On at least one system we're running,* `ps -haemer` *works fine.) If so, what output do you get?*

For AIX, neither `ps -haemer` (which works on BSD systems) nor `ps -copeland` works: Both result in bad flag errors. On the other hand, `ps -jeff` gives a full process listing.

Jeffrey Copeland (copeland@alumni.caltech.edu) *is currently looking for interesting work in a project-management or senior-technical capacity. His reasearch interests include internaionaliza-tion and typesetting. He lives in Austin, TX, where he raises children, cats and roses.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

• *Why would* `ctermid()` *return* `/dev/tty` *instead of something useful?*

You can communicate with the controlling terminal, if there is one, by writing to `/dev/tty`, so it makes sense for `ctermid()` to return that, if it exists.

Onward to the Beach

Let's consider the sketchy program in Listing 1. Notice that since this is a toy version, we've not been careful about several things. For example, we are not particular about the exit code from the routine `sh()`. Also, if we haven't defined the environment variable `PS1`, there is no default command prompt. Similarly, if `HOME` is undefined, the behavior of `cd` will not be good.

Let's continue to consider `cd` for a moment: Why is it handled as a separate case in the routine `built_in()`? Certain operations have to be handled internally to the shell, not by external programs. Changing directories is one of them. If we did the `cd` in a separate program, that program would be running in a different directory, while this invocation of the shell would remain where we started it.

Let's review the actions in `command()`: We talked about the `fork()` and `exec()` interfaces several columns ago, and now we're seeing it in action. We parse the input line into its blank delimited arguments, and if `built_in()` returns zero, we `fork()`. This creates a new process, duplicating the existing one. Remember that `fork()` returns a 0 to the child process–that is the new process–and returns the process id of the child to the original process. The child process then does an `execvp()` to invoke the command line. The parent process invokes a `wait()`, which returns when the child terminates.

Exercise for the reader: How does our `command()` routine differ from the standard `system()` interface?

We are not done with the shell yet. We haven't done anything about I/O redirection, or pipes, which means our favorite command line:

```
find . -type f -print | xargs egrep Jeff
```

won't work. We are missing other built-in functions, such as read and for. (How would you make the current directory part of the `PS1` string?) Environment variables aren't parsed on the command line. We need to handle invoking commands in background. The exit code for the shell needs to be defined and cleaned up. We can't fix all of

Listing 1

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/limits.h>
#include <stdlib.h>
#include <string.h>

main(int argc, char *argv[])
{
  /* argument processing */
  exit(sh());
}

sh()
{
  char *cp;
  int command(char *);
  char *PS1 = getenv("PS1");
  char s[_POSIX_MAX_CANON];

  for (;;) {
    fputs(PS1, stdout);
    if ((cp = fgets(s, _POSIX_MAX_CANON, stdin)) == NULL)
      return(0);
    s[strlen(cp)-1] = '\0';
    command(s);
  }
}

command(char *s)
{
  char *args[_POSIX_ARG_MAX];
  int built_in(char **);
  char **cpp = args;
  char *IFS=" ";
  int stat, pid;

  for (*cpp = strtok(s, IFS); *cpp != NULL;
    *cpp = strtok(NULL, IFS))
    cpp++;
  if ((!*args) || built_in(args))
    return;
  if ((pid = fork()) = = -1)
    err_sys("can't fork");
  if (pid = = 0) {
  execvp(args[0], args);
    exit(1);
  }
  else
    wait(&stat);
}
built_in(char **args)
{
  char *PWD;
  if (strcmp(args[0], "cd") = = 0) {
    PWD = args[1] ? args[1] : getenv("HOME");
    chdir(PWD);
    return(1);
  }
  else
    ; /* etc. */
  return(0);
}
```

these, or even the missing features from the POSIX shell we *haven't* mentioned, but we can explore some of them.

Background Jobs

Let's begin by fixing our shell to invoke commands in the background. To do that, we add two things: We need to recognize the & at the end of the command line, and we need to not wait for the command to complete. We begin by adding two declarations at the top of the program:

```
#include <string.h>
int background;

main( ... )
```

Then we look for the ampersand, by changing:

```
if ((cp = fgets(s, _POSIX_MAX_CANON, stdin)) = = NULL)
   return(0);
s[strlen(cp)-1] = '\0 ';
```

to this:

```
if ((cp = fgets(s, _POSIX_MAX_CANON, stdin)) = = NULL)
   return(0);
bg = 0;
if (cp = strrchr(s, '&'))
   ++bg;
else
   cp = s + strlen(s) - 1;
*cp = '\0 ';
```

Now, we can change our action after the fork() in command():

```
if (pid = = 0) {
   execvp(args[0], args);
   exit(1);
} else if (bg)
   return(0);
else
   waitpid(pid, &stat, 0);
```

We now use waitpid(), which allows us to wait for a specific process. Why? Because wait() returns when *any* of the children terminates. Consider the following scenario:

```
$ wc /unix &
$ cc -o shell shell.c
$ shell
```

If the wc completed before the cc, then the wait() would allow command() to return and collect the last command line, shell, which would fail because the compilation wasn't complete yet. (Pretty obvious, right? We had to think about it for a minute before we got it correct.)

Let's also add a built-in to let the shell wait if we want it to by adding this to built_in() at the *etc* comment:

```
else if (strcmp(args[0], "wait") = = 0)
     wait(&stat);
```

This allows us to reconsider the & we added to a command line and wait for the command instead. Two points to ponder, before we move on: Does wait() need to be a system call? Or can it be a library function? Also, does the wait command need to be built in?

I/O Redirection

Being able to redirect standard input and output is one of the simple notions in UNIX that has turned out to be so powerful. The redirection is handled by the shell. But how? This is where the dup() interface comes in handy. We'll demonstrate by including output redirection in our shell. Let's begin by adding the relevant declarations:

```
#include <sys/stat.h>
#define MODE_666
  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
int fd;
```

(Notice that we've invented a shorthand for the file permissions we want.)
Next, we need to check for an output file name, in command().

```
if (out = strrchr(s, '>')) {
        *(out++) = ' ';
        out = strtok(out, IFS);
}
```

Lastly, we need to connect the standard output to the file:

```
if (pid = = 0) {
  if (out) {
    fd = creat(out, MODE_666);
    dup2(fd, 1);
  }
  execvp(args[0], args);
  exit(1);
}
```

Notice that we've actually used the dup2() interface, which closes the file associated with the second argument (in this case, standard output), and then assigns the file associated with the first argument (in this case, the file we just opened), to that file descriptor. So, our new file is associated with file descriptor 1, so any write to standard output goes to the file named in out. dup() and related interfaces act across exec() calls, which allows us to set up the files before we invoke the new program. The POSIX standard tells us that we don't need `dup()` or `dup2()` (even though they are both provided in Standard C), with the `F_DUPFD` subcommand. That is, `dup(fd)` is equivalent to `fcntl(fd,F_DUPFD,0)` and `dup2(fd,fd2)` is the same as `close(fd2); fcntl(fd,F_DUPFD,fd2)`. The `dup2()` interface has not always existed, being implemented as macros in many places where the functionality was needed. Why bother with a separate routine then? Because it provides us with a clean, common interface, and the library routine as specified in Standard C does some argument checking for us.

Timing our Commands

Some time ago, we played with the `time` command, and the related `times()` interface. As you'll remember, `times()` takes a pointer to a time accounting buffer. That buffer contains four things: the user CPU time used by the children of this process, `tms_cutime`; the system CPU time used by the children of this process, `tms_cstime`; the user CPU time used by this process *and* its children, `tms_utime`; and the system CPU time used by this process and its children, `tms_stime`. The time for the children recorded after a `wait()` would return when the child process ends. Also, `times()` returns the elapsed real time relative to an arbitrary point in the past. These times are all measured in clock ticks per second, `CLK_TCK`.

In some shells, `time` is a built-in command, but it's a pretty simple program and can be written as a stand-alone, as we'll do in Listing 2.

Listing 2

```
#include <sys/types.h>
#include <sys/times.h>
#include <stdio.h>
#include <time.h>

main(int argc, char *argv[])
{
  struct tms tbuf;
  pid_t pid;
  clock_t start, end;
  int stat;

  if (argc < 2)
    err_quit("usage: %s command", argv[0]);
  if ((pid = fork()) = = -1)
    err_sys("can't fork");
  start = times(&tbuf);
  if (pid = = 0) {                    /* child */
    execvp(argv[1], argv + 1);
    err_sys("exec failed");
  } else {                            /* parent */
    pid = wait(&stat);
    end = times(&tbuf);
    fprintf(stderr, " -12s%3.2f ,
      "real", (double)(end - start)/CLK_TCK);
    fprintf(stderr, "%-12s%3.2f ,
      "user", (double)(tbuf.tms_cutime)/CLK_TCK);
    fprintf(stderr, "%-12s%3.2f ,
      "system", (double)(tbuf.tms_cstime)/CLK_TCK);
    exit(0);
  }
}
```

Points to note:

• We invoke `times()` twice. This first time merely gives us the starting real time, and we ignore the `tbuf` it returns.

• Why don't we check the error return on `execvp()` and just always print the error message? Because if an `exec()` returns, the new program has failed to start up. If the call was successful, it never returns, and the new image exits.

• Similarly, the `else` clause is unnecessary. If we are in the branch of the `fork` where `pid` is 0, then we will either successfully `exec()` the new program, and terminate from there, or exit on failure from the child.

Exercises for the Reader

This month, our back-of-the-envelope exercises are programming problems. To wit:

• We've ignored command stacking in which POSIX allows lines such as:

```
$ date; ls
$ sleep 10 & date
$ sed 's/foo/bar/' <zzazz >mumble && mv mumble zzazz
```

How would you implement these? Does this change how you'd implement shell return code handling?

• We've written output redirection. How would you add input redirection? Slightly trickier: What about standard error?

• We've also written output redirection in a way that allows commands of the form:

```
$ sed "s/UNIX/Unix/g" in >out
```

but not:

```
$ sed "s/UNIX/Unix/g" >out in
```

How can we fix this? How does this affect your changes for input redirection?

That's it for our visit to the beach. Next time we'll visit the clock factory and further explore how we tell time on POSIX systems. From there we'll segue into a discussion of signals and interprocess communications. Until then! ▲

In Which We Check Out the Clock Factory

by Jeffreys Copeland and Haemer

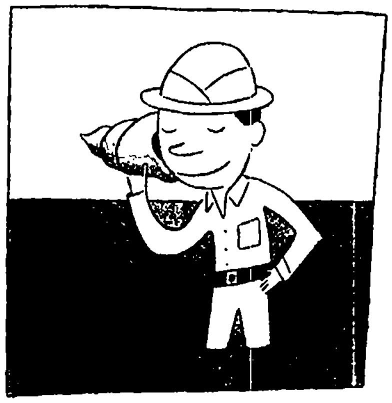
Last month, we did some exploration of fork() and exec() and built a version of the shell. This month, we'll revise that version of the shell and explore the POSIX time-keeping functions.

The Shell, Revisited

Let's start with last month's exercises for the reader. Normally, the questions we end the column with are what Donald Knuth would classify as 10 or 15. (If you aren't familiar with the logarithmic scale used for the exercises in *The Art of Computer Programming*, it ranges from 0 for immediate, such as "what's the sum of the first three positive integers," to 50 for a research problem, such as Fermat's last theorem. It turns out to be a useful way of expressing the difficulty of problems in real life, too.) This time, we left you with problems that involve a bit of coding and rank about 20 or 25. We left you with three problems:

- *We've ignored command stacking, in which POSIX allows lines such as:*

```
$ date; ls
$ sleep 10 & date
$ sed 's/foo/bar/' <zzazz
>mumble && mv mumble zzazz
```



How would you implement these? Does this change how you'd implement shell return code handling?

- *We've written output redirection. How would you add input redirection? Slightly trickier: What about standard error?*
- *We've also written output redirection in a way that allows commands of the form:*

```
$ sed "s/UNIX/Unix/g" in >out
```

but not

```
$ sed "s/UNIX/Unix/g" >out in
```

[This wasn't strictly true, but we'll touch on that here, too.] How can we

Jeffrey Copeland (copeland@alumni.caltech.edu) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

fix this? How does this affect your changes for input redirection?

Rather than spend the whole column writing code, we'll ignore the first problem and solve the second two in one lump. Holding that thought in mind, if you look at Listing 1, you'll find the unedited shell code we had at the end of last month's column.

We can handle the changes for all three kinds of I/O redirection and the argument order with just changes to the `command()` routine.

Let's begin by adding some declarations:

```
char *in = NULL, *out = NULL, *err = NULL, *t;
```

We need the initializations because we won't be using the variables in a way that automatically sets them. In addition, we want to change the way we parse the arguments into the `args[]` array. This involves changing the initial `for` loop to:

```
for (t = *cpp = strtok(s, IFS); *cpp != NULL;
    t = *cpp = strtok(NULL, IFS)) {
   if( *t = = '<' )
      in = t;
   else if( *t = = '>' )
      out = t;
   else if( strncmp(t, "2>") )
      err = t;
   else
      cpp++;
}
```

Now, at the end of the argument parsing loop, each of the file name pointers is either NULL or points at the redirection symbol. We need to extract the file name, so we replace the current `if( out = strrchr(s, ...` clause `command()` with the following:

```
if( in ) {
 in++;  /* skip past redirection symbol */
 in = strtok( in, IFS );
}
if( out ) {
 out++; /* skip past redirection symbol */
 out = strtok( out, IFS ); }
if( err ) {
 err += 2; /* skip past "2>" */
 err = strtok( err, IFS );
}
```

Now all that remains is to expand the file opening after the `fork()`. To do this, we add some code inside of the `if( pid = = 0 )` statement.

```
if (pid = = 0) {
 if (in) {
   fd = open(in);
   dup2(fd, 0);
 }
 if (out) {
   fd = creat(out, MODE_666);
   dup2(fd, 1);
 }
 if (err) {
   fd = creat(err, MODE_666);
   dup2(fd, 2);
 }
 execvp(args[0], args);
 exit(1)
}
```

Listing 1

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/limits.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#define MODE_666
  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
int fd;
int background;
main(int argc, char *argv[])
{
    /* argument processing */
    exit(sh());
}
sh()
{
    char *cp;
    int command(char *);
    char *PS1 = getenv("PS1");
    char s[_POSIX_MAX_CANON];
    for (;;) {
      fputs(PS1, stdout);
      if ((cp = fgets(s, _POSIX_MAX_CANON, stdin)) = = NULL)
        return(0);
      if( cp = strrchr(s,'&'))
        ++background;
      else {
        cp = s + strlen(s) - 1;
        *cp = ' ';
      }
      command(s);
    }
}
command(char *s)
{
    char *args[_POSIX_ARG_MAX];
    int built_in(char **);
    char **cpp = args;
```

Notice that we still have a small problem here. We correctly handle

```
cat foo >bar
```

but not

```
cat foo > bar
```

The extra space just causes us to have an empty file name. This is why: In addition to a long discussion of shell command syntax and semantics, POSIX.2 specifies the grammar for the shell in the form of a yacc source. Given the yacc grammar for the parser of our toy shell, it would be much easier to add the command stacking we're going to ignore here.

```
    char *IFS="  ";
    char *out;
    int stat, pid, fd;
    for (*cpp = strtok(s, IFS); *cpp != NULL;
         *cpp = strtok(NULL, IFS))
       cpp++;
    if ((!*args) || built_in(args))
       return;
    if (out = strrchr(s, '>')) {
       *(out++) = ' ';
       out = strtok(out, IFS);
    }
    if ((pid = fork()) = = -1)
       err_sys("can't fork");
    if (pid = = 0) {
       if (out) {
          fd = creat(out, MODE_666);
          dup2(fd, 1);
       }
       execvp(args[0], args);
       exit(1);
    } else if( background )
       return( 0 );
    else
       wait(&stat);
}
built_in(char **args)
{
    char *PWD;
    if (strcmp(args[0], "cd") = = 0) {
       PWD = args[1] ? args[1] : getenv("HOME");
       chdir(PWD);
       return(1);
    } else if (strcmp(args[0], "wait") = = 0 )
       wait( &stat );
    else
       ;        /* etc. */
    return(0);
}
```

If Mickey's Little Hand Is on the Four...

We ended our discussion of shell internal functions last month with the time command. As you'll recall, this tells us how much user, system and clock time our process used. To pull off this trick, we used the times() interface, which returns the elapsed time relative to an arbitrary time in the past. (What time? We really don't care, since we are using it to determine elapsed time, and as you'll recall from your high school algebra class, if x y z, then x y c=z c.)

That's all well and good, but how do we tell what time it really is on the wall clock? How does the date appear on the banner page on the printer? How does the date command know what time it is and what time zone I live in? We've already discussed some of these issues in the context of file access times in our sixth, seventh and eighth columns. Let's discuss them now in terms of how POSIX tells clock time.

POSIX provides three sets of time information. The first set, which we discussed last time, is the information on command timing, definitions for which appear in <sys/times.h>. Times for file access are defined in <utime.h>, which we discussed when we covered files. Lastly, Standard C defines <time.h>, which covers the wall clock time, and which defines a flock of interfaces, such as asctime() and strftime(). Consider the following program:

```
#define _POSIX_SOURCE
#include <time.h>
main()
{
  time_t t;
  t = time((time_t *) NULL);
  printf("At the tone, the time will be %s ,
    ctime(&t));
}
```

(What does the \a in the printf() format string do? It's the same as the ASCII character \007; that is, it rings the bell. But, as we've been preaching all along, \a is preferred because it's portable, while \007 works only in ASCII.)

In this program, we first call time(), which gives us the number of seconds since the epoch–midnight GMT, 1 January 1970. For the truly interested, page 200 of the POSIX.1 standard provides a full-page rationale for use of the epoch, since the original definition was considered too loose. For example, "seconds since midnight GMT, January 1, 1970" doesn't specify what happens for leap seconds. (They're ignored.)

Note that time() both returns the current time and puts it in the location we send as an argument, unless the argument is NULL, as in our example; in that case, the time is simply returned. (Why does time() act this way? Why not just return the time_t value of the current time? This is another example of POSIX.1 codifying existing practice. Initially, the C language didn't provide a long data type, so time() couldn't return its value. Instead, the routine was invoked with the address of a two-integer array, into which was stored the halves of the long specifying the time.)

Returning to our example program: We translate the number of seconds returned by time() to an ASCII string using the ctime() interface, which is defined in Standard C. This routine returns a pointer to a character string such as Wed Jul 13 16:44:11 1994.

Two other useful interfaces for time are localtime() and gmtime(), which take the time_t we get from time() and break it down into a struct tm, which contains the times in Figure 1.

Figure 1

```
int tm_sec;     /* seconds after the minute - [0,59] */
int tm_min;     /* minutes after the hour - [0,59] */
int tm_hour;    /* hour since midnight - [0,23] */
int tm_mday;    /* day of the month - [1,31] */
int tm_mon;     /* months since January - [0,11] */
int tm_year;    /* years since 1900 */
int tm_wday;    /* days since Sunday - [0,6] */
int tm_yday;    /* days since Jan 1 - [0,365] */
int tm_isdst;   /* flag for daylight savings time */
```

What's the difference between the struct tm returned by gmtime() and that returned by localtime()? The first returns the Coordinated Universal Time, and the second returns the local time.

How do we determine the local time? From the TZ environment variable. Traditionally, this has had a value such as CST6CDT in Austin, or MST7MDT in Boulder. (Notice that time zones west of the meridian–that is, earlier than UTC–have positive offsets. This is just the reverse of what you'd expect if you were designing a system from the ground up, but it's another artifact of the standard codifying existing practice for a system originally developed in North America.) POSIX now defines TZ to be of the form *std offset [dst [offset]] [,rule]* where *rule* is *start[/time],end[/time]*.

Pretty obscure, right? Well, *std* and *dst* are the names of the standard and summer time zones. Each has an offset from Coordinated Universal Time, of the form *hh[:mm[:ss]]*. In places such as Japan, where there is no

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
Features:	High	Medium	Low
Taking the Plunge into Objects	170	171	172
Putting Objects in Place	173	174	175
Columns:			
Q&AIX–File Systems on a Diet	176	177	178
Systems Wrangler–AIX System Parameters	179	180	181
Datagrams–The Recent History of the Internet	182	183	184
AIXtensions–DCE Time and Again	185	186	187
POSIX–We Check Out the Clock Factory	188	189	190

daylight savings time, only the first part is used. (Why both minutes and seconds? Because there are places such as Iran, Afghanistan, India and central Australia that have time zones on the half hour rather than the even hour. Up until the 1970s, the time in Singapore was GMT+7:45.) The *rule* is the complicated part and tells us when summer time begins and ends. For the United States, the rule is: `M4.1.0/2:00:00,M10.5.0/2:00:00`. That is: begin daylight savings time on the first Sunday of April at 2 a.m., and end it on the last Sunday of October at 2 a.m.

What is POSIX.1 lacking in the way of time functions? We don't have an explicit way to set the time–that's provided by POSIX.2 in the `date` command. We don't have a `gettimeofday()` function, to just give us clock time. We don't have any functions to do a sanity check on the values we feed into a `struct tm`.

Our personal favorite time function remains `strftime(buf,fmt,tm)`. This takes a `struct tm` and fills a buffer with the formatted time specified in a `printf`-style `fmt` string. For example, the two fragments:

```
char buf[128], *s = &buf[0];
time_t t;
t = time( (time_t *) NULL );
s = ctime( &t );
```

and

```
char buf[128], *s = &buf[0];
time_t t;
t = time( (time_t *) NULL );
strftime( s, "%a %b %d %T %Y%n", &t );
```

fill `buf` with the same value. There are roughly three dozen specifiers for `strftime()`, many of which apply to non-English language, and non-Western calendars. We discussed this interface in some detail in our series of columns on internationalization.

Until Next Time...

We leave you with one multipart exercise: What non-Western calendars should we handle in a routine like `strftime()`? How many of them are practical to handle? (We'll give this a Knuth rating of 10.) Instead of folding these into `strftime()`, choose one calendar, and write the language-specific equivalent of `asctime()` for it. (We'll give that one a Knuth rating of 25.)

So Long for Now

We didn't manage to get to our expected discussion of signals and interprocess communication, but we'll begin with that next time. In the meantime, keep those cards and letters coming, folks. ▲

HOW EASILY CAN SOMEONE GAIN ACCESS TO YOUR SYSTEM WITHOUT AUTHORIZATION?

CAN YOU:

Set an automatic profile expiration date?
Require regular password changes?
Prevent the re-use of old passwords?
Control the time and location for all user logins?
Keep a detailed audit log of all account changes?
Keep a log of all valid and invalid login attempts?
Force password changes after a security breach?

If the answer to any of these questions is 'no', then you need GUARDIAN, the account and access control system with a proven world-wide reputation for enhancing Unix system security.

DataLynx, Inc., 6659 Convoy Ct. San Diego, CA 92111 **(619)-560-8112 Fax: (619)-560-8114**
Le Software Man, P.O. Box 545 London, N7 8DF, U.K. **(071)-354-8414 Fax: (071)-226-2015**

Circle No. 8 on Inquiry Card

WE BUY, SELL & RENT

RS/6000

PARTS • FEATURES
UPGRADES • CPU'S

• System 36 Conversions
• AutoCad Available for RS/6000
• Sun Sparc Compatibles

New & Used IN STOCK
Complete Technical Center, Installation,
Stock Parts & Features for RISC.

COMPUTER MARKETPLACE
(800) 858-1144

A Publicly Traded Company NASDAQ: MKPL

***MOTOROLA UDS** Western Regional Distributor of the Year*

CDLA IBM®

1490 RAILROAD ST., CORONA CA 91720
TEL (909) 735-2102 • FAX (909) 735-5717

Circle No. 6 on Inquiry Card

In Which We Discover Signals: The Other IPC

by Jeffreys Copeland and Haemer

Last month, we dropped an assortment of calendar questions with idiosyncratic answers in your laps. Answering the most instructive of these–how to write a language-specific equivalent of `asctime`–will take a while, so we'll begin with...

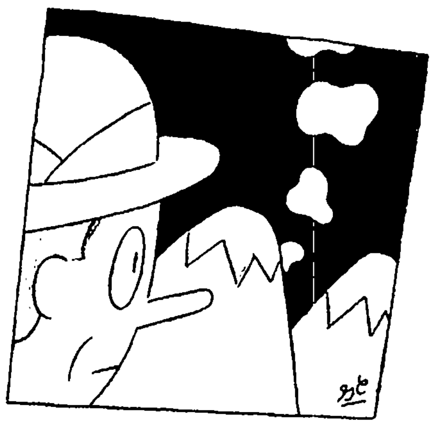
What's that? We've been promising to talk about signals and you want to hear about them instead? Well, we'll get to them right after...

Eh? You'd rather hear about them right now? OK, OK. But I wish you'd stop interrupting.

Exceptions

All operating systems have to detect and address exceptional events–hardware failures at the very least. (Not often, of course; otherwise, they wouldn't be exceptions.) In the simplest imaginable case, the operating system might detect the exception and just terminate all running processes. Most operating systems do better than this and let user-level programs die gracefully, closing open files and so on. In some cases, it's even possible to specify a way to recover cleanly from an exception–discarding a bad result, or notifying the user and restarting from some synchronization point.

The user-level code called when an exception occurs is an exception handler. Think of this as code that's never called explicitly but is asynchronously branched to, as if with a "go to," when some outside condition occurs. The condition can occur any time, between any two statements, within a statement or while executing a system call. Some languages, such as Ada, have long provided portable constructs to handle exceptions. Traditional (Kernighan and Ritchie) C, did not, but we'll return to that in a moment.



Software Interrupts

When else might we want to arbitrarily interrupt the flow of a program? A runaway program? Control-

Jeffrey Copeland (copeland@alumni.caltech.edu) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

C, Delete, Break or Control-backslash usually does the trick, even if you haven't put in code between each line to check whether the user has typed one of these keys. How does this work? UNIX has always handled what are known as software interrupts by generalizing the idea of exception handlers. The operating system detects and notifies applications of user- or application-generated signals just as though they were hardware exceptions. Therefore, we approach writing an interrupt handler for `SIGQUIT` the same way we do for `SIGFPE` (floating point exception)–with Control-backslash. If you've lived in UNIX all your life, you may take this for granted.

'Tain't so. One of us remembers working on a port of a large UNIX application to a proprietary operating system that had only exception handling. Figuring out how to simulate UNIX signal handling was a major part of the port. Although traditional C lacks any way to specify interrupt handlers, exception handlers are ubiquitous, and standard C (that is, ISO/ANSI C) uses the traditional UNIX `signal()` function to let users tie user-level interrupt handling code to particular signals. In the prototype, as you'd guess, the other half of the problem–user-generated signals–is a touch trickier, since not all operating systems let user-level processes generate arbitrary signals. The traditional UNIX mechanism is kill(), which sends a signal from one process to another. The prototype is just

```
int kill(pid_t pid, int sig);
```

On single-process systems, this would be overkill(), so standard C provides `raise`, which lets a process send a signal to itself. This is portable to any system that has a standard C compiler, but its sharply limited utility makes it seem like looking under the lamppost for your keys.

(A man sees a drunk on his hands and knees underneath a lamppost, scouring the ground for something.

"What did you lose?"

"I've dropped my keysh."

"On the street side, or the sidewalk side?" the man asks, getting down on his knees to help.

"Actually, I dropped them about half a block back, but the light's a lot better here.")

One of the UNIX systems we've used has the line:

```
#define raise(x) kill(getpid(), x)
```

in <signal.h>.

It seems to us as though `signal()` should mean "signal a process," and "handle a signal" should be called `handle()`, or something, but POSIX follows historical practice, which is UNIX historical practice.

Interprocess Communication

Really, then, although signal processing started as a generalization of hardware exception handling, it's more useful to think of signals on UNIX systems as interprocess communication (IPC) mechanisms. Like most UNIX systems, AIX has a largish array of such mechanisms, including shared memory, semaphores, sockets and message queues, but the only ones covered by POSIX.1 are those from System III UNIX: pipes (together with their close kin, FIFOs) and signals. We've spent some time talking about pipes; now we'll spend a little talking about "signals: the other POSIX IPC."

Unfortunately, signals have traditionally had two problems as IPC mechanisms: limited capabilities and reliability.

The limited capabilities of signals come from their very small bandwidth. Realistically, the small number of possible signals that an application can respond to limits the number of things you can communicate to an application through a signal. Signals are also limited because you shouldn't do too much inside a signal handler. If you start getting signals when inside a signal handler, control flow within applications built around using signals for interprocess communication would become confusing. A good rule of thumb is "don't do much inside signal handlers": save a little state, set some flags, close a file or two if you have to, but return quickly to where you left off. The reliability problem was one of bad design. In early UNIX systems, entering a signal handler reset the signal to its default behavior; to catch a signal received while in the signal handler, you had to put `signal()` statements at the top of your signal-handling code. Berkeley UNIX fixed this by changing the semantics of signals, but there's so much code out there that assumes the traditional, unreliable `signal()` semantics that standard C's `signal()` function leaves its reliability unspecified. POSIX solves the reliability problem by supplying a new set of interfaces for reliable signal handling. The semantics are roughly the same as those of Berkeley signals.

Old code containing calls to `signal()` will still work and will be as unreliable as it ever was, but new code should use the new interfaces. We won't talk about `signal()`. Don't use it.

What? You want to talk about last month's exercise? Now? Another interruption? (sigh) All right.

Asctime() Revisited

Obviously, we could write a whole flock of `asctime()` equivalents for different languages that printed "6 Juin 1944," for example, instead of "June 6, 1944." If you read our previous series on internationalization (*RS/Magazine*, May 1992-April 1993), you know that this trick is correctly handled with a locale definition, which contains, among other information, the names of the months in various languages. Our real question was, for what calendars, other than Gregorian, would it be useful to have a special version of asctime()? As you probably realize, the standard Gregorian civil calendar isn't the only one used in the world. The Russian Orthodox Church still operates on the Julian calendar; the Hebrew and Islamic calendars are the time lines for secular and religious life in communities worldwide (ever try cashing a check on Friday in Riyadh?); years are numbered differently in Japan; there are separate calendar schemes for Chinese and Indian cultures; decoding the Mayan calendar allowed scholars to understand their mathematics. Given all that, let's write two different versions of asctime(), one for the Japanese date and one for the Islamic calendar.

Let's start with the Japanese calendar, which is simpler. In Japan, the Gregorian calendar is used with a minor variation: The years are numbered based on the year of the current emperor's reign. For example, the day we mark as August 8, 1991, would be *Heisei 3 nen 8 gatsu 20 nichi*, the 20th day of the eighth month in the third year in the era of Emperor Akihito. Note that Japanese time spans are based on one, not zero, so a newborn baby is one year old. As a result, 1989, the year in which Hirohito died and Akihito took the throne, was not only *Heisei 1 nen*, the first year of Heisei, but also *Showa 64 nen*, the 64th year of the Showa era, or the reign of Hirohito. So, given a tm structure, we can implement ja_asctime as shown in Listing 1.

Listing 1.

```
char *era_name[ ] = { "Heisei", "Showa", "Taisho", "Meiji", 0 };

ja_asctime( tm )
struct tm *tm;
{
  int era, year;

  if( later( tm, 1989, 1, 7 ) ) {
    era = 0;
    year = tm->tm_year - 1989 + 1;
  } else if( later( tm, 1926, 12, 24 ) ) {
    era = 1;
    year = tm->tm_year - 1926 + 1;
  } else if( later( tm, 1912, 7, 29 ) ) {
    era = 2;
    year = tm->tm_year - 1912 + 1;
  } else if( later( tm, 1868, 9, 7 ) ) {
    era = 3;
    year = tm->tm_year - 1868 + 1;
  } else {
    error( "the year %d is too early for a known emperor\n",
                     tm->tm_year );
    return 1;
  }

  printf( "%s %d nen %d gatsu %d nichi\n",
    year, tm->tm_mon, tm->tm_day );
  return 0;
}

/* return true if the m/d/y given is later than the
   date in the tm structure */
later( tm, y, m, d )
struct tm *tm;
int y, m, d;
{
  if( tm->tm_year < y )
    return 1;
  else if( tm->tm_mon < m )
    return 1;
  else if( tm->tm_day < d )
    return 1;
  return 0;
}
```

Note that we really could use the value in tm_yday to check the first day of the eras, instead of writing the utility routine later(), but the code we've written is clearer. Also, in a real implementation we would have printed our results with the appropriate kanji, or Japanese characters. Lastly, UNIX time doesn't extend earlier than *Showa* 45 (that's 1970 to us *gaijin*), so much of our code is unnecessary.

On the other hand, the Islamic calendar...Yes, you in the back row waving the semaphore flags. You had an interrupt? You want to talk about POSIX signals instead? All right.

POSIX Signal Handling

The basic idea to get in your head about POSIX signals is that they come in sets, of type sigset_t. You can imagine implementing a signal set as an array, as bits in a word, or as a C++ class, or even as a linked list–POSIX doesn't say. For us, a signal set is an abstract data type, with the following POSIX.1-specified operators:

- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
- `int sigaddset(sigset_t *set, int signo);`
- `int sigdelset(sigset_t *set, int signo);`
- `int sigmember(const sigset_t *set, int signo);`

Each of these does what it looks like it does. For example, `sigaddset()` adds a particular signal to a set, and so on. None of these operators affect signal handling or delivery in any way, but many other POSIX signal-related functions expect signal sets as arguments, rather than individual signal numbers. Sets come in handy from time to time and, in theory, there's no reason we couldn't use variables of type `sigset_t`, together with these operators, when the occasion arises. The elements, however, need to be legitimate signal values, for the operators to be guaranteed to work. The set of available signals varies from implementation to implementation, but these signals are guaranteed to be available: SIGABRT, SIGALRM, SIFGPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2.

On systems with the job control option (`#ifdef _POSIX_JOB_CONTROL`), you're also guaranteed these: SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU. We'll return to why job control is special later on, but nowadays almost every UNIX operating system (including AIX) supports job control because the U.S. government's procurement regulations, set out in the National Institute of Science and Technology's FIPS 151-2, require it.

Don't Just Stand There, Do Something

Having defined signal sets, what can we do with them? The first, and most trivial, is to block signals and to ask what signals are pending. The functions for this are

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

and

```
int sigpending(sigset_t *set);
```

Traditional UNIX signals couldn't be blocked. These interfaces and functionality are new.

Traditional signal handling is also available, but is now set up with a call to the function

```
int sigaction(int sig, const struct sigaction *act,
   struct sigaction *oact);
```

The first argument is the signal to be handled (note the singular: "the signal"). The second and third are the new sigaction structure and the value of the sigaction structure before the call.

"What the heck," you may be asking yourself, "is a *sigaction* structure?" Here's what it looks like:

```
struct sigaction {
   void (*sa_handler)(); /* signal handler */
   sigset_t sa_mask; /* added during handling */
   int sa_flags; /* only for SIGCHLD */
   /* can have other members */
};
```

The first field is the name of the function that is the signal handler. These can be the names of user-defined functions, or can be the special values SIG_DFL or SIG_IGN. Just as with the old `signal()` interface, SIG_IGN says "ignore the signal," and SIG_DFL says "restore default handling of the signal." The second field is more interesting. When processing a signal handler set up with `sigaction()`, the signal being processed is automatically blocked. For example, if I'm an application that has a signal handler for SIGQUIT, when I enter the signal handler, any future SIGQUIT signals are blocked until I return.

What the sa_mask lets the programmer do is specify an additional array of signals to block within the signal handler. Often, this field is set to contain an empty set, but it need not be. The third field is more abstruse; you probably won't need it unless you write shells. We'll leave its exploration to the bold-at-heart.

Don't Just Do Something, Stand There

The pair of functions `kill()` and `sigaction()` do nearly everything normally involved in signaling and signal handling, but there are a handful of ancillary functions worth mentioning.

One of these is the traditional `int pause()`, a Snow White-like function that puts the process into a deep sleep, broken only by the kiss of a princely signal. (Can you tell we both have 9-year-old daughters?) Processes that are paused will awaken to any signal that the process hasn't blocked.

If you'd prefer being more selective about the signals you'll respond to, use `sigsuspend()`. This function takes a signal set as an argument, which are the signals to block receipt of. When `sigsuspend()` returns, the previous signal mask is restored. (You might be thinking that you could do the same job by combining `sigprocmask()` and `pause()`, but you'd be overlooking the possibility that a signal might arise between the two calls. The `sigsuspend()` function is atomic.)

The function `sleep()` puts your process to sleep for a specified amount of time, or until the process receives a signal, whichever comes first. The related function `alarm()` sets an alarm clock, which generates the alarm signal `SIGALRM` after a certain number of seconds.

Because the `read()` function blocks until it completes, a traditional way to write code that waits for user input, but not forever, is to call `alarm()` before calling `read()`; the `SIGALRM` will break out of the `read()` if no input is available, and let the application either continue, or retry the `read()`. It's not hard to imagine writing `sleep()` with `pause()` and `alarm()`. Indeed, the standard counsels that the two calls are so closely related that your application should use only one or the other, not both.

Deep Trivia

Because we've run out of space, we'll defer discussing the remainder of the topic of signals (and giving you problems to think about) until next time. And if you promise not to interrupt, we'll finish talking about the Islamic calendar and write an `isl_asctime()`.

We'll wind up this installment by digressing, without apology or real justification, into our favorite piece of POSIX trivia: the functions `sigsetjmp()` and `siglongjmp()`.

Standard C incorporates the two traditional UNIX functions that collaborate to offer nonlocal `goto`s: `setjmp()` and `longjmp()`. The first of these, `setjmp()`, marks a spot in your code; the second, `longjmp()`, returns you to it. (A `goto` usually won't work for this because it is only permitted to go to a point within the same function.)

These two functions find frequent use in signal handlers. Signals can happen anywhere, and sometimes the safest thing to do after getting a signal that may be deep within a maze of logic is to return to some safe, well-specified spot in the code and start afresh. This can be done by doing a `setjmp()` at the synchronization point, and then doing a `longjmp()` from the signal handler. Ah, but here's the catch. What's the signal mask after the `longjmp()`? Is it the mask set in the signal handler? The mask set just before the signal handler was called? The mask before the `setjmp()`? Without knowing the mask, it could be hard to write code that behaved reproducibly and reliably.

If the second argument to `sigsetjmp()` is nonzero, a return to that point also resets the signal mask to the one in force at the time that `sigsetjmp` was called. (The `siglongjmp()` call is only needed because you can only return to `sigsetjmp()` with a `siglongjmp()`.) Until next time, folks. ▲

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located elsewhere in this magazine. Rate the following column and feature topics in this issue.

	INTEREST LEVEL		
Features:	High	Medium	Low
The High Road to Parallel Processing	170	171	172
Look What's Listening	173	174	175
Columns:			
Q&AIX–Beyond Archie	176	177	178
Systems Wrangler–A Command Named Su	179	180	181
Datagrams–Bastion Hosts, Firewalls and Socks	182	183	184
AIXtensions–UNIX to Scale	185	186	187
POSIX–Signals: The Other IPC	188	189	190

In Which We Discuss the Environment

by Jeffreys Copeland and Haemer

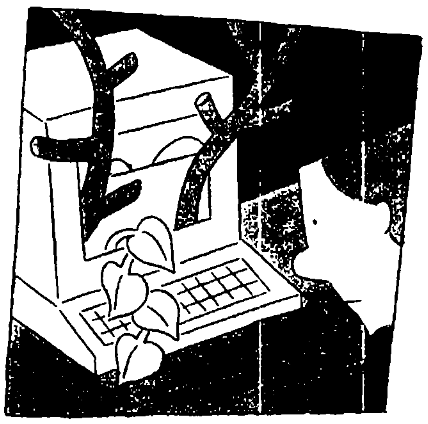
This month, we want to discuss water quality and the state of the rain forest. We'll spend a little time talking about...What? Oh! Sorry, wrong environment.

We actually want to talk about process environments. That is, about the information that surrounds your program in your POSIX system. But first, we want to finish answering a question we posed two months ago: How would you write a version of `asctime()` for a non-Western calendar? Last month, we showed you how to handle Japanese Imperial dates, and this month we'll finally show you our code for `isl_asctime()`, which handles Islamic calendars.

If It's Ramadan, This Must Be Riyadh

Remember when you were in college, and every once in a while one of your profs would say, "Let's spend the day discussing something that's not in the syllabus"? Well, that's more or less what we're going to do for the first half of this month's column.

We started this discussion as a toss-off question about calendars two months ago. We originally asked about interesting non-Western calendars for which to implement `asctime()`. When it came time to implement the code, we wanted to do it for a lunar calendar. In practical terms, that meant either the Hebrew or the Islamic calendar, because the Chinese and Mayan ones would take multiple columns to discuss. It turns out that the Islamic calendar is a little more straightforward than the Hebrew one. Why? Because the Hebrew calendar has very funny leap year rules so that Passover, which



Jeffrey Copeland (copeland@alumni.caltech.edu) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

replaced an early planting festival, remains in the spring. (Interesting side note: Because both Easter and Passover are tied to a calculated full moon, they are either within a week of each other or a month apart.) Even using the Islamic calendar, the easier of the two, the algorithms are complicated, and it took a lot of effort to get the code right. So, let's put the soundtrack from *Lawrence of Arabia* on the stereo and pass the bowl of dates while we show you how the Islamic calendar works.

First, credit where credit is due: We've mentioned them before, but Nachum Dershowitz and Edward M. Reingold at the University of Illinois have done a wonderful study of calendric calculations, and it is from their work that we have cribbed our algorithms. (Check out their paper "Calendric Calculations" in *Software–Practice and Experience*, 20 [9] [September 1990], pp. 899-928.)

The Muslim calendar is strictly lunar, so it consists of 12 months alternating between 30 and 29 days. Leap years don't follow a four-year cycle but occur in 11 out of every 30 years. And in a leap year, the last month has 30, rather than 29, days. Let's begin by writing a pair of utility routines to tell us the number of days in a month, and whether we have a leap year (see Listing 1).

Listing 1

```
/* What's an Islamic leap year?
   One whose Islamic year mod 30
   is 2, 5, 7, 10, 13, 16, 18,
   21, 24, 26, or 29 */
int
isl_leap( int year )
{
   switch( year ) {
   case 2: case 5: case 7: case 10:
   case 13: case 16: case 18: case 21:
   case 24: case 26: case 29:
      return 1;
   default:
      return 0;
   }
}

/* How many days in a month?
   This takes an *Islamic* month
   and year as arguments!! */
int
isl_mon_len( int mon, int year )
{
   if( mon & 01 )
      return 30;
   else if( isl_leap(year) && mon = = 12 )
      return 30;
   else
      return 29;
}
```

We will also find it useful to have the names of those months available:

```
char *isl_mon_name[] = { "",
   "Muharram", "Safar",
   "Rabi I", "Rabi II",
   "Jumada I", "Jumada II",
   "Rajab", "Sha'ban",
   "Ramadan", "Shawwal",
   "Dhu al-Qada", "Dhu al-Hijjah" };
```

Dershowitz and Reingold use a trick that we've used ourselves: They number days from an arbitrary point in the past. (Let's call these absolute day numbers.) They choose to use the first of January, 1 A.D., on the Gregorian calendar, as a starting point. Never mind that there wasn't a Gregorian calendar two millenia ago; it turns out to be a convenient starting point because, among other reasons, it was a Monday. So, let's write a quick routine to give us the day number of a given Gregorian date (see Listing 2).

Now we need to know something about when the Islamic era started. It began with the Hegira, Mohammed's flight to Medina, on the (Julian) 16th of July, 622. If the Gregorian calendar had existed, that would have been the 19th of July, 622. Using `abs_greg_date()` above, we discover that the absolute day number of the beginning of the Islamic calendar is 227015, which is 1 Muharram 1 A.H. (*anno hegira*). From the information so far, we can write a routine to give us the absolute day number from the Islamic date:

Listing 2

```
int greg_leap_year( int year );
int days_in_greg_mon[] =
  { 0, 31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31 };

long
abs_greg_date( int mon, int day, int year )
{
  unsigned long sum;
  int i;

  /* Begin with days so far this year: */
  for( sum = 0L, i = 1; i < mon; i++ )
    sum += (long) days_in_greg_mon[i];

  /* Plus days in current mon: */
  sum += (long) day;

  /* Plus regular days in previous years */
  sum += (long) ((year - 1) * 365L);

  /* Plus leap day this year? */


  if( mon > 2 &&
    greg_leap_year(year) )
      sum++;

  /* plus previous leap days */
  sum += (long) ((year - 1) / 4L);
  sum -= (long) ((year - 1) / 100L);
  sum += (long) ((year - 1) / 400L);

  return sum;
}
```

Of course, the calculation of leap years is bunches easier:

```
int greg_leap_year( int year )
{
  if( (year % 400) = = 0 ) return 1;
  if( (year % 100) = = 0 ) return 0;
  if( (year % 4) = = 0  ) return 1;
  return 0;
}
```

```
#define ISLAMIC0 227015L

long
abs_isl_date(
    int mon, int day, int year )
{
  long sum;
  sum = ((mon-1) * 29L)
       + (mon / 2L) + day;
  sum += (year-1) * 354L;
  sum += ((year * 11L) + 3L) / 30L;
  sum += ISLAMIC0 - 1L;
  return sum;
}
```

That's all the setup we need. Now we can write a routine to take a Gregorian date and print out the Islamic one.

```
isl_print_date( int mon, int day, int year )
{
  long abs_date;
  int isl_year, isl_mon, isl_day;

  /* What's the absolute Gregorian
    date we're asking about? */
  abs_date =
    abs_greg_date( mon, day, year );
  printf( "%d/%d/%d (%ld) -> ",
    mon, day, year, abs_date );

  /* If the date is before 1/1/1 AH,
    forget it. */
  if( abs_date < ISLAMIC0 )
  {
    printf( "error: this date is "
      "before Islamic calendar began );
    return -1;
  }
```

That part was easy. We could do a very complicated calculation to get the Islamic month, day and year, but we choose to take a guess and then do a linear search. We begin by making a lower bounds estimate of the year, and then using the abs_isl_date() routine above to close in on the correct year.

```
  /* Lower bound on year */
  isl_year = (abs_date - ISLAMIC0 - 1L) / 355L;

  /* Search for the actual year */
  while( abs_isl_date(1,1,isl_year+1)
         <= abs_date
    isl_year++; }
```

Similarly, we home in on the month:

```
for( isl_mon = 1;
    abs_isl_date(isl_mon,
         isl_mon_len(isl_mon,isl_year),
         isl_year)       < abs_date;
    isl_mon++ )
     continue;
```

Then, the correct day of the month is obtained by subtraction, and we're done.

```
isl_day = abs_date
    - abs_isl_date(isl_mon, 1, isl_year) + 1;

/* print the result */
printf( "%s %d, %d A.H. ,
         isl_mon_name[isl_mon],
    isl_day, isl_year );
return 0;
```

Given all that, we can exercise our code with the {co main ()} routine in Listing 3, and we get the output in Listing 4.

Please note that we haven't actually written an `isl_asctime()`, which takes a `struct tm *` and emits an ASCII string. With the realization that this is exactly how we started this digression, we leave that last step as an exercise for the reader.

There is a major caveat: This calendar is calculated. Strictly speaking, some Muslims wait until the new moon has been proclaimed by religious authorities before changing the month. As a result, your computed first of Shawwal may differ from the one recognized by your more pious neighbor.

Back to Our Regularly Scheduled Topic

What does POSIX.1 say about environments? What useful information can I get about who is running my program? What about the computer my program is running on? What about the characteristics of the system

Listing 3

```
main( )
{
  isl_print_date( 1, 1, 1 );
  printf("\t 7/19/622 is 227015 or Muharram 1, 1 \n");
  isl_print_date( 7, 19, 622 );
  printf("\t 11/12/1945 is 710347 or Dhu al-Hijjah 6, 1364 \n");
  isl_print_date( 11, 12, 1945 );
  printf("\t a date of no significance in Arabia \n");
  isl_print_date( 7, 6, 1776 );
  printf("\t note that we can have the same day twice in a Gregorian year \n");
  isl_print_date( 1, 5, 1930 );
  isl_print_date( 12, 25, 1930 );
}
```

Listing 4

```
1/1/1 (1) ->      error: this date is before Islamic calendar began
      7/19/622 is 227015 or Muharram 1, 1
7/19/622 (227015) ->      Muharram 1, 1 A.H.
      11/12/1945 is 710347 or Dhu al-Hijjah 6, 1364
11/12/1945 (710347) ->    Dhu al-Hijjah 6, 1364 A.H.
      a date of no significance in Arabia
7/6/1776 (648493) ->      Jumada I 19, 1190 A.H.
      note that we can have the same day twice in a Gregorian year
1/5/1930 (704557) ->      Sha'ban 4, 1348 A.H.
12/25/1930 (704911) ->    Sha'ban 4, 1349 A.H.
}
```

running on that computer? Most of these questions are actually asking *where* your process is, for arbitrary values of "where."

Let's try an example to begin our exploration. We can find out who you are with the following short program:

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <unistd.h>
main(void)
{
  puts(getlogin());
}
```

Around our offices this program often provides the output jeff for arbitrary values of "jeff."

Historically, UNIX provides a way of getting information about the system you're running, with the uname command. If you've been paying attention for the last 14 months, you've guessed that POSIX.1 has a system interface to get this information for you. And you'd be right.

The uname() call populates the utsname structure, defined in the <sys/utsname.h> include file. The utsname structure contains at least the name of the implementation of the operating system (sysname), name of the UUCP node (nodename), current release level of the system (release), current version level of the system

If you've been paying attention for the last 14 months, you've guessed that there is a system interface in POSIX.1 to get this information for you. And you'd be right.

Table 1

Attribute	Type	Get Function	Set Function
process ID	pid_t	getpid()	–
parent process ID	pid_t	getppid()	–
process group ID	pid_t	getpgrp()	setpgid()
login user name	char*	getlogin()	–
real user ID	uid_t	getuid()	setuid()
effective user ID	uid_t	geteuid()	setuid()
real group ID	gid_t	getgid()	setgid()
supplementary group IDs	gid_t[]	getgroups()	–
current directory	char*	getcwd()	chdir()
file mode creation mask	mode_t	umask()	umask()
signal mask	sigset_t	sigprocmask()	sigprocmask()
set of pending signals	sigset_t	sigpending()	–
process times	struct tms	times()	–
controlling terminal	char*	ctermid()	–
session ID	–	–	setsid()
password file entries	struct passwd*	getpwuid()	–
group file entries	struct group*	getgrid()	–

(`version`) and name of the hardware (`machine`). If you wrote a version of the `uname` program, using the `uname()` interface, you might find out from an RS/6000 near you that:

- its operating system is called `AIX`
- the system is at Release 2 of Version 3
- the machine type code is 75, i.e., a 7015 Model 375
- and in the case of a machine near us, the nodename is `armadillo`.

We've written a lot of code already this month, so we'll leave `uname` as an exercise for the reader.

This process carries around quite a lot of information about the environment. It's summarized in Table 1.

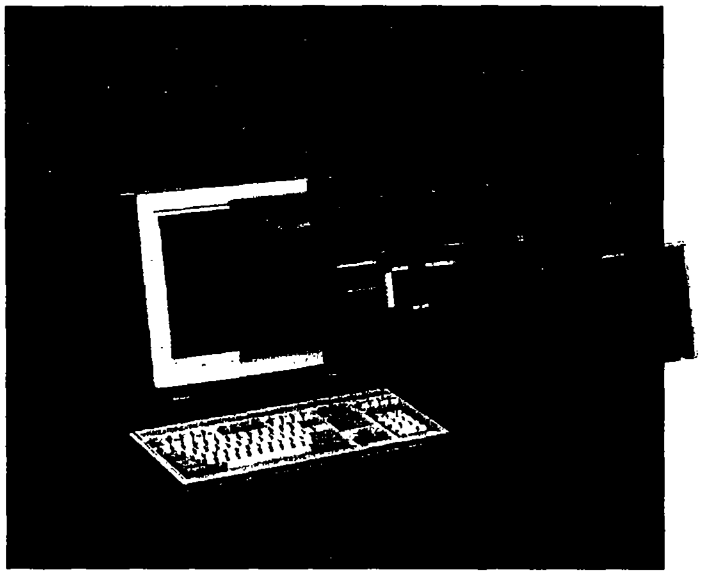
You'll notice that you can set some of this information, too. This is how the `cd` built into the shell works, for example. As for those arbitrary values of "where," you can even get arbitrary named information out of the process space with the `getenv()` interface.

If you will remember, the point of POSIX was to codify existing practice, not to provide a new standard for an imaginary operating system. To do this, though, it was necessary to provide the occasional innovation. For example, how can you codify the amount of space needed for a file name? Do you specify it as 14 bytes, as in traditional AT&T UNIX, in arbitrarily long, as in Berkeley derivatives? Why not, instead, allow you to ask the system at runtime? That is the point of the POSIX.1 interfaces `sysconf()` and `pathconf()`. To find the number of time intervals per second, try `sysconf (_SC_CLK_TCK)`. For finding out the length of a file name, try `pathconf (".",_PC_PATH_MAX)`. (Why do you need to provide a file name to `pathconf()`? With network file systems, and remote mounting, it is possible that a file system configuration parameter may vary within a single host.) In general, `sysconf()` provides information about the general system configuration, and `pathconf()` tells you about parameters associated with files and data movement. A full table of the information from these two calls is provided in the POSIX.1 standard in tables 4-2 and 5-2. Some of it may also be found in the `include` file `limits.h`.

Wrapping Up

That's it for information about the environment, albeit information that's a little compressed since we rearranged the syllabus. Tune in next month when we discuss device-specific functionality, and visit the dreaded `tcgetattr()` interface. Until then! ▲

Facet*Term* Version 3: Your Window On Unlimited Terminal Productivity



Increase your productivity with the Facet*Term*® windowing environment on character terminals, AlphaWindow terminals and PCs.

Facet*Term* features:

- Run up to 10 applications simultaneously
- Use <u>any</u> character application on <u>any</u> UNIX system with <u>any</u> character terminal
- Cut and paste between applications
- Get automatic notification of Email and any other "off screen" activity
- Print screens
- Utilize new AlphaWindow terminals

Free Evaluation! Full features, documentation and support for 30 days.

Structured Software Solutions, Inc.

Opening Windows on Terminal Productivity

Structured Software Solutions, Inc.
4031 West Plano Parkway
Plano, Texas 75093
TEL: (800) 235-9901 (214) 985-9901
FAX: (800) 982-9901 (214) 612-2035
Email: info@sssi.com

OUTSTANDING PRODUCT

SCO PRODUCT OF THE YEAR 1993
System Utility Software
SCO

Facet*Term* is a registered Trademark of Structured Software Solutions, Inc.
Other names are properties of their respective holders.

Circle No. 18 on Inquiry Card

In Which We Discuss Replacements for the ioctl() System Call

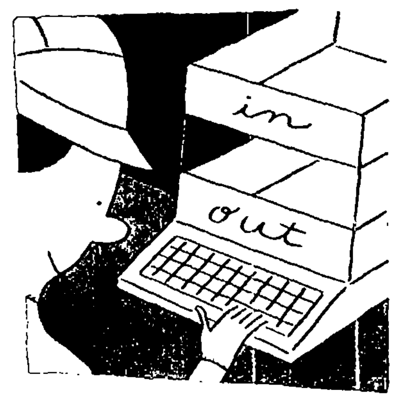
by Jeffreys Copeland and Haemer

This month, in our continuing exploration of the POSIX.1 standard, we will discuss the changes POSIX has made in getting data in and out of your terminal.

The Ubiquitous Backspace

Let's begin by considering what happens when you type Control-H on your keyboard at the shell prompt. (If you read our series on internationalization, you've seen some of this discussion before.) The Control-H doesn't appear. In fact, something entirely different happens: The cursor moves backward, and the previous character disappears. This is because the standard input to the terminal is in "cooked" (or "canonical") mode: The terminal driver is processing the characters before it passes them on to the shell.

What do you do if you want to process the Control-H character yourself? (We realize that a more realistic example would be processing the escape codes generated by the arrow keys so that you could navigate around a dungeon. For the purposes of this column, however, we're going to explore a simpler example.) Historically, you would have used some form of the `ioctl()` interface, which existed as far back as Version 7 UNIX, to put the terminal into "raw" mode. Unfortunately, `ioctl()` was intended to provide a general I/O control facility, and nearly each driver for each device relied on a different number of arguments to `ioctl()`, of different types–try to provide an ANSI C pro-



Jeffrey Copeland (copeland@alumni.caltech.edu) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

totype for that! Indeed, between Version 7 and System III releases of UNIX, the arguments to ioctl() changed entirely. Worse, the Berkeley UNIX releases provided an entirely different set of arguments and subcommands for the interface and added hooks for governing features like job control.

This is another case of the POSIX.1 standard not quite following existing practice. In this case, instead of adopting either the BSD or USG styles of terminal control, POSIX adopted something that' s fairly close to the System III termio structure, called termios. To communicate with the termios structure, we have two interfaces: tcgetattr(int filedes, struct termios *tp), which allows us to get information about a terminal, and tcsetattr(int filedes, int options, const struct termios *tp), which sets it. This means, in general, that we write programs of the form

```
tcgetattr(fd, &otbuf);
ntbuf = otbuf;
/* modify ntbuf for new settings */
tcsetattr(fd, 0, &ntbuf);
/* application program continues with new settings */
tcsetattr(fd, 0, &otbuf);     /* cleanup */
/* now terminal is reset to original settings */
```

If you've ever aborted out of, say, vi and left the terminal in a funny state, you can probably guess that the part you failed to execute is the line with comment "cleanup."

It might be useful at this point to take a look at the contents of the termios structure:

```
struct termios {
  tcflag_t    c_iflag;      /* Input modes */
  tcflag_t    c_oflag;      /* Output modes */
  tcflag_t    c_cflag;      /* Control modes */
  tcflag_t    c_lflag;      /* Local modes */
  cc_t        c_cc[NCCS];   /* Control characters */
};
```

Each of the tcflag_t elements contains a set of flags. The array of control characters in cc_cc[] allows us to remap characters for interrupt and erase. There may be other fields in the structure, but these are the ones POSIX guarantees.

Magic Procedures

Let's try a simple example. (No, we're not going to process the arrow keys this time–we'll leave that as an exercise for you.) We'll simply echo the characters we read from the keyboard and investigate the differences between "raw" and "cooked" mode. We can start by setting up the main program (see Listing 1).

Notice that we do as we claimed. We echo all the characters we read (expanding control characters), except that when we read an "x" we invoke setraw() or restore(). What do these magic procedures do? Let's do the simple one first (see Listing 2).

We need to take a closer look at those middle lines in setraw(), beginning with setting tbuf.c_iflag, which actually do the work. First, we unset some input flags:

- We don't want to translate the NL character to CR, and vice versa (INLCR | ICRNL).
- We don't want to strip the input characters to 7 bits.
- We actually want to see the START and STOP characters rather than use them to implement flow control (those characters are defined in c_cc, which we'll return to).
- We want to prevent a break condition at the terminal (generally, this means you pressed the Break key) from flushing the terminal driver's input and output queues.

On the other hand, the output flags are much simpler. There is only one defined by POSIX, and we are unsetting it–though there may be some others in your implementation that we are leaving alone. Turning off OPOST prevents any postprocessing of the output characters on their way to the terminal screen. Finally, we reset three bits in the local flags to prevent canonical processing. That is, don't automatically process the erase (normally Control-H) and kill (Control-U) characters, but instead pass them on to the program to disable signals, so we don't automatically process the interrupt and quit characters (normally Control-C and Control-backslash). Most important, turn off echo, so that our program is solely reponsible for displaying the characters we read. Notice that we have done nothing with the c_cflag bits, which control line conditions like number of bits per byte and number of stop bits. See Figure 1 for a complete list of the bits corresponding to each element of the flag set.

Finishing It Up

To complete things, we do some setup in the control characters array. The VMIN and VTIME values in the c_cc array are useful only when we are reading in noncanonical mode. For canonical input, the read() call returns when a new line is received. But, in canonical mode, the read() (which means the getchar(), too) completes after buffering VMIN characters. So, if we set VMIN to 1, we can act after each character is typed.

What about VTIME? If VTIME is zero, we return when we've read VMIN bytes. On the other hand, if VTIME is greater than zero, we return when either we have VMIN characters, or VTIME seconds have passed without a character being detected.

Why? Because noisy serial lines are common. The following is a list of other control characters we can manipulate:

VEOF	end of file character
VEOL	end of line character
VTIME	time to wait before returning an error
VMIN	minimum characters to read
VERASE	erase character (normally Control-H)
VINTR	interrupt character (Control-C)
VKILL	kill character (Control-U)
VQUIT	quit character (Control-backslash)
VSUSP	suspend character (Control-Z)
VSTART	start character (Control-S)
VSTOP	stop character (Control-Q)

Figure 1. The Flag Set

Field	Mask	Description
c_iflag	BRKINT	interrupt on break
	IGNBRK	ignore break condition
	IGNCR	ignore CR character
	IGNPAR	ignore characters with parity errors
	INPCK	enable input parity check
	PARMRK	mark parity errors
	ICRNL	map CR to NL on input
	INLCR	map NL to CR on input
	ISTRIP	strip character to 7 bits
	IXOFF	enable flow control on input
	IXON	enable flow control on output
c_oflag	OPOST	perform implementation-dependent output processing
c_lflag	ECHO	enable local echo
	ECHOE	echo ERASE as backspace-space-backspace
	ECHOK	echo KILL character
	ECHONL	echo \n
	ICANON	canonical input
	IEXTEN	enable extended (implementation defined) functions
	ISIG	enable signals
	NOFLSH	don't flush buffers on INT, QUIT or SUSPEND
	TOSTOP	send SIGTTOU for background output
c_cflag	CLOCAL	ignore modem status lines
	CREAD	enable receiver–otherwise read no characters
	CSIZE	number of bits per byte
	CS5	5 bits
	CS6	6 bits
	CS7	7 bits
	CS8	8 bits
	CSTOPB	send two stop bits
	HUPCL	hang up on exit
	PARENB	enable parity
	PARODD	odd parity

So Where's ioctl()?

If we don't have `ioctl()` anymore, how do we handle line control, baud rate and other device control functions? What do we end up missing?

Last questions first: The main thing we lose is Berkeley-style word erase. POSIX doesn't support it at all. Among other problems, how do you define word erase for a language like Japanese where words are not delimited by white space? (Note that the Berkeley Control-W operator works on AIX, anyway. For Japanese text, it deletes back to the last ASCII white-space character.) Furthermore, we lose compatibility with existing implementations. To go back to the arrow key example, to port your dungeon game from BSD to a strictly POSIX platform, you'd now need to change all the terminal setup code. To provide the other terminal functions we get from `ioctl()`, POSIX provides roughly a dozen routines:

• `tcsendbreak()`	send a break
• `tcdrain()`	drain the input queue
• `tcflush()`	flush the input queue
• `tcflow()`	set flow control on the input queue
• `cfgetospeed()`	get the output baud rate
• `cfsetospeed()`	set the output baud rate
• `cfgetispeed()`	get the input baud rate
• `cfsetispeed()`	set the input baud rate
• `ctermid()`	get the terminal identification
• `ttyname()`	get the name of the terminal
• `isatty()`	tell us if we are really on a terminal

What about the `ioctl` coverage for job control? What about other devices? For job control, there are POSIX routines such as `tcgetpgrp()`, which gets the group ID of the foreground process group.

Listing 1

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <termios.h>

struct termios tbufsave;
void setraw(void), restore(void);
int raw;

main()
{
    int c;

    while(1) {
          c = getchar();
          if (c == EOF)
               break;
          if (iscntrl(c)) {
               putchar('^');
               putchar(c + 0x40);
          } else if (c == 'x') {
               if (raw)
                    restore();
               else
                    setraw();
          } else
               putchar(c);
    }
}
```

Listing 2

```
void restore(void)
{
    raw = 0;
    if (tcsetattr(0, TCSANOW, &tbufsave) == -1) {
          fprintf(stderr, "tcsetattr failed );
          exit(1);
    }
}

void setraw(void)
{
    struct termios tbuf;

    raw = 1;
    if (tcgetattr(0, &tbuf) == -1) {
          fprintf(stderr, "tcsetattr failed );
          exit(1);
    }
    tbufsave = tbuf;
    tbuf.c_iflag &= ~(INLCR|ICRNL|ISTRIP|IXON|BRKINT);
    tbuf.c_oflag &= ~OPOST;
    tbuf.c_lflag &= ~(ICANON|ISIG|ECHO);
    tbuf.c_cc[VMIN] = 5;
    tbuf.c_cc[VTIME] = 2;
    if (tcsetattr(0, TCSANOW, &tbuf) == -1) {
          fprintf(stderr, "tcsetattr failed );
          exit(1);
    }
}
```

As for other devices, POSIX provides the `fcntl()` interface. However, note that there's no attempt to make the actions of `fcntl()` parallel to the functions provided by the terminal controls we've just described. How does it work? `int fcntl( int fildes, int cmd, ... )` takes a file descriptor and a command to execute. The commands are defined in `<cfntl.h>` and include commands like `F_DUPFD`, which is similar to the function of the `fdup()` interface; `F_GETFL`, which returns the status flags of the open file; or `F_GETLK`, which checks an advisory lock on the file.

Advisory locking is probably the most interesting of the three. It's called "advisory" because it's not enforced. Another program only knows that you have locked part of a file by calling `fcntl()` and checking existing locks.

And how does it work? In general, we request a lock with the `F_GETLK` command and a pointer to an flock structure. The flock structure contains the type of lock requested (read or write), where in the file we want to lock, and how large a region of the file we want to lock. We check a lock with `F_GETLK`, which returns type `F_UNLCK` if there is no existing lock. We can then request a lock with `F_SETLK` or `F_SETLKW`; the latter waits until the file is unlocked to return. When we are done, we set the lock type to `F_UNLCK` and call `fcntl()` again. All in all, it's pretty straightforward, as long as everyone plays by the same rules and provides functionality that wasn't standard in Version 7 or System III.

The Days Dwindle Down...

We've now taken the simple notions of terminal interface and file control and beaten them to death. Next month, we will consider some odds and ends, such as the standardized format for `tar` (...or was it `cpio`?) archives. Following that, we'll sum up our programmer's view of POSIX.1. ▲

In Which We Discuss Odds and Ends

by Jeffreys Copeland and Haemer

Well, we're about at the end of our survey of POSIX.1-1990. We've shown you its general structure and principles, and talked in some detail about the two big areas encompassed by the standard: file systems and processes. Before we finish, we need to touch on a few loose ends that are in the tail end of the standard.

A Data-Exchange Standard

The purpose of the POSIX standards is to provide application portability across a wide range of machine types, all running UNIX-like operating systems. Seems sensible enough. But if you're an old hand at porting, you'll know that just moving the bits from one machine to another can be a challenge. We'll admit that it's better now than it used to be–one of us remembers working on CDC6400s, running Kronos, where you had a choice of ASCII character sets. Any time someone gives you "a choice of ASCIIs," you know you're in trouble. Still, the diversity of physical media and data formats can make your boss wonder why it's taking so long to move your "portable" programs. The POSIX.1 committee decided it had to address this question and sort of did.

As you'd expect, venturing into the territory of physical media was out of the question. This is a software standard designed to buffer the programmer from the speed of hardware evolution. In 1988, when the original standard passed, no one had CD-ROM drives, people still used 5½-inch floppies and DAT tapes were out of everyone's reach. Neverthe-

Jeffrey Copeland (copeland@alumni.caltech.edu) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

less, the committee did do something almost as surprising: They decided to specify a data format.

It's worth pausing here to note how big a step this was. All of the POSIX standards are interface standards. A reluctance to specify data formats and data structures is deeply woven into software standards aimed at portability. Data abstraction–separating the interfaces from the underlying data representation–is what lets programs written to the POSIX C standard on a PC running MS-DOS, compile and run on an MVS machine with an EBCDIC character set. Likewise, because POSIX.1 is careful to avoid specifying the directory's format, the marketplace now offers POSIX.1-conforming MVS, VMS, CTOS and MP/E systems. You can move a carefully written, POSIX.1-conforming application from one to another with a recompile, despite the fact that the underlying file systems and process structures vary wildly. Although the POSIX standards are clearly modeled on UNIX, there is no working group standardizing an equivalent to the traditional section seven of the manual: file formats.

Usually we don't think of standards organizations as bold, but the decision to specify a data format in an interface standard was genuinely gutsy.

Tar Wars

We have been careful to say that the committee decided to specify a data interchange standard instead of saying that they agreed on one. Having agreed to stipulate that POSIX.1 systems would support a data interchange format, the first candidate to step up was cpio. This was because the 1984 /usr/group standard was based on AT&T's System III (remember when the phone company owned UNIX?).

Fortunately, hotter heads prevailed, and people quickly began appearing at the committee meetings who pointed out that cpio was inadequate for the vast number of BSD-based systems. They also said that BSD-based systems had as much right to the sacred label of "existing practice" as AT&T-based systems.

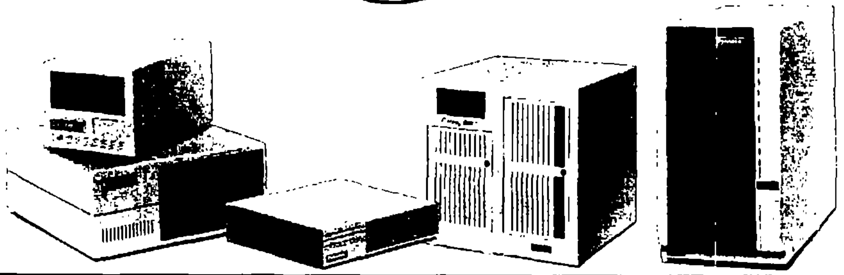
Why inadequate? The cpio command will take a list of UNIX file names and archive the corresponding files into a single file, which may be put out on a external medium for transport. Conversely, it will also take a single archive and restore some or all of the files. This means it has to know about file names and file types. And it does: System III file names and System III file types. The data structures can't handle long file names, symbolic links or sockets or symbolic user and group names. (Believe it or not, bin is not always group 7 on all systems.)

"Ah!" you say. "But that's where tar comes in." As it turns out, tar falls equally short in some subtler areas. One solution to the quandary would have been to fix one of the two formats. Unfortunately, each utility has half a zillion (we counted) shell scripts in the field that call it, and a similar number of religious adherents. 3CPIO's and TAR2-D2's resulting fierce struggle for dominance was, of course, christened "tar wars."

The deadlock was eventually broken by Glenn Fowler, who came to a meeting armed with an overhead that asked a simple question: Which format file is this?

```
07070700200701461510066400014400002400000
0206563405633121233000021000000011357odd
s_and_ends.mm^@. " $Id$
.S 12
.TL
POSIX From the Outside In:
.sp
A Retrospective
.AF ""
.AU "Jeffreys Copeland & Haemer" "" ""
.MT 4 1
```

The Gig is UP & Prices DOWN.



1.2 Gb Disk **$860**	1.9 Gb Disk **$1,320**	2.4 Gb Disk **$1,605**	3.5 Gb Disk **$2,192**	5.0 Gb Disk **$2,575**
10.8 Gb Disk **$4,205**	2.0 Gb 4 mm **$1,025**	4.0 Gb 4 mm **$1,120**	5.0 Gb 8 mm **$1,510**	10 Gb 8 mm **$2,295**

Prices are for fully assembled & tested subsystems with enclosure, cable and terminator

A complete family of mass storage products for RS/6000, SUN, HP, SGI, DEC, & NOVELL.

Dynamic Computer Products...not just price, but over 25 years of technical expertise, service and support.

Dynamic COMPUTER PRODUCTS

233 Greenwood Avenue, Bethel, Connecticut 06801

Tel: 203/791-9511 • 1-800-714-SCSI • Fax: 203/778-4535

Circle No. 12 on Inquiry Card

At this, the shouting stopped. It turns out that almost none of the formats' rabid adherents actually knew what the formats looked like. Glenn then announced that he not only knew the difference but could distinguish the two on the fly, which meant that a single utility could be constructed that would read (or write) either format.

The committee then agreed to require that POSIX.1 systems support both formats. They also specified backward-compatible extensions to each format to solve the problems identified during the debates, and the Usenix association funded the development of a publicly available utility, called pax ("peace," in Latin) that would read or write either, demonstrating that Fowler was right.

The cpio format is unimaginatively called "extended cpio," but the extended tar format is called ustar and pronounced either "U.S. tar" or "u-star." You'll find all three on your system. On many systems, though not on AIX, an ls -l will reveal that the three commands are links. Usenix's pax is designed to behave like cpio when called by that name, like ustar when called by that name, and to use its own flags and format when called as pax. (Try the experiment to see if each format will work on your system. On AIX, however, the three programs are distinct.)

Last but Not Least

Section 8 of POSIX.1 has the peculiar title "Language-Specific Services for the C Programming Language." This seems odd for a few reasons.

First, the entire standard is a C-language binding. A pair of standards exist that bind POSIX.1 services to other languages–POSIX.5 for Ada and POSIX.9 for FORTRAN-77–that required considerable effort to work around the language-specific features utilized by POSIX.1. For example, three of the POSIX.1 directory entry routines, readdir(), rewinddir() and closedir(), all take pointers to dirent structures as arguments, and the fourth, opendir(), returns the pointer. FORTRAN77 has neither structs nor pointers.

Section 8 carefully connects C's stdio functions to POSIX.1's low-level I/O functions.

Second, standards are, in theory, orthogonal. The entire point of separating out the C and POSIX.1 standards is that even a non-POSIX operating system like MS-DOS should be able to support a standard C compiler. Indeed, we'll remind you that POSIX.1 doesn't require standard C at all. Although the C in the standard itself looks like standard C, it's legal to have either a standard-C binding or a traditional (K&R) C binding–support for prototypes, for example, isn't required. Because that's true, the POSIX.1 standard should have a clean boundary that separates the language from its operating system-specific functions.

'Tain't so.

Here's why. First, although standard C isn't required, POSIX.1 does require that even traditional C implementations provide a specific subset of the functions in the standard. Section 8 sets these out, and the subset is large, but not all-inclusive. For example, although setlocale() is required, the functions that deal with multibyte characters, mbtowc() and friends, are not.

Second, POSIX.1 has to extend the semantics of a handful of the standard C functions: setlocale(), rename(), abort(), getenv(), ctime(), gmtime(), localtime(), mktime() and strftime().

Let's take an example. The call rename("/usr", "/usr/bin") is illegal in POSIX.1 because /usr/bin is a subdirectory of /usr. To prohibit this sort of thing, standard C would have to know about directories and pathnames, which it doesn't.

Some restrictions are relaxed, too. POSIX.1 disavows any distinction between text files and binary files. This means that on proprietary systems that *do* make such distinctions, POSIX.1-conformance can require some fancy footwork.

Another example is setlocale(), which knows about the LC_ environment variables. Standard C knows about environments (getenv()) but specifies no environment variable names, because not all operating systems have environment variables.

Third, POSIX.1 has to connect some of its functions to C-language functions. One of the topics we covered in our treatment of file systems was low-level I/O. The UNIX programmer's assumption is that the standard I/O library is implemented through calls to open(), close(), read(), write() and seek(), but the C-standard specification of the functions in <stdio.h> makes no such statements. On an MS-DOS machine, for example, the underlying BIOS calls need not be anything like the UNIX operations. Section 8 carefully connects C's stdio functions to POSIX.1's low-level I/O functions. For example, POSIX.1 requires that fclose() do a close() on the underlying file descriptor and update the st_ctime and st_mtime fields of the file.

In fact, the standard even provides

a pair of interfaces to interconvert POSIX.1 file descriptors and C `FILE` pointers: `int fileno(FILE *stream)` and `FILE *fdopen(int filedes, const char *type)`. The first of these takes a `FILE` pointer and returns the associated file descriptor, and the second takes a file descriptor and returns a `FILE` pointer that you can use to do input or output to the file using standard I/O functions.

Another Day, Another Standard

Well, that's about it for POSIX.1-1990, and just in the nick of time! The IEEE has now passed IEEE 1003.1b-1993, which has the 10 chapters we've covered so far, plus five more. These five chapters, plus extensions and modifications to the first 10, add "real-time" interfaces to POSIX.1.

Although we noted the existence of a few POSIX options in earlier columns, such as `_POSIX_JOB_CONTROL` and `_POSIX_CHOWN_RESTRICTED`, all of the interfaces we've discussed so far have been mandatory. This is not true of the interfaces in the newer chapters, whose presence is under the control of a wasp's nest of new POSIX options. For example, if `_POSIX_SEMAPHORES` is true, then the implementation supplies the process synchronization interfaces described in Section 11. If not, then whether they're there or not is up for grabs. If they're there, however, they must behave as described in the standard. Here are the new chapters and what they supply:

• **Section 11: Synchronization**–Interfaces to create and use named and unnamed semaphores (`_POSIX_SEMAPHORES`). For those who had to suffer through System V semaphores, these aren't the same.

• **Section 12: Memory Management**–Three relatively disjointed sets of interfaces: memory locking (`_POSIX_MEMLOCK` and `_POSIX_MEMLOCK_RANGE`), shared memory (`_POSIX_SHARED_MEMORY_OBJECTS`) and memory mapped files (`_POSIX_MEMLOCK`). Memory locking lets real-time processes lock mapped pages into memory, for performance enhancement. Shared memory is much like the System V model, and memory-mapped files are done with Berkeley's `mmap()`. Although `mmap()` can provide shared memory facilities, the standard has a separate set of shared-memory routines for systems that want to supply shared memory without implementing a full-blown mechanism for memory-mapped files.

• **Section 13: Execution Scheduling**–This section has interfaces that let you set process priorities and scheduling policies on a per-process basis (`_POSIX_PRIORITY_SCHEDULING`). You can choose fifo (`SCHED_FIFO`), round-robin (`SCHED_RR`) or an implementation-specific (`SCHED_OTHER`) scheduling algorithm, set priorities and even yield the processor.

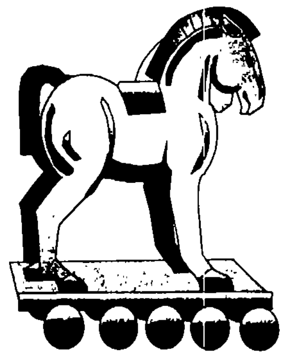
• **Section 14: Clocks and Timers**–This section supplies per-process timers and high-resolution sleeps (`_POSIX_TIMERS`). Real-time processes are characterized not so much by speed as by predictability. You can tell a real-time system by the fact that its response times are given as ranges instead of averages. Sections 13 and 14 combine to help provide that predictability.

• **Section 15: Message Passing**–Not System V message queues, but similar enough to lack surprises (`POSIX_MESSAGE_PASSING`).

We won't say any more about these new chapters in this series for a simple reason: We don't know them well enough (yet). If, however, you're an expert in this area, we suspect that the folks here at *RS/Magazine* would love to hear from you.

That's all for us for the moment. Next month, we'll be back for the grand finale: a review of the original POSIX.1. ▲

TROJAN HORSE
in your stables?



Last year trojan horses compromised tens of thousands of user accounts and passwords on the Internet.

Don't let this happen to you.

STOP
Internet Intruders
with
PORTUS™
a secure firewall
system from
LSLI™

CALL 1-800-240-5754
FAX 1-713-379-5225

PORTUS was developed at the IBM Thomas J. Watson Research Center to secure its private network when connected to the Internet and has been in production since 1988. PORTUS is based on the current version of this secure gateway and represents the state of the art in securing a network from intruders.

Contact **LSLI** to secure your gateway with the finest software technology in the world ... PORTUS.

© 1994 LSLI

PORTUS and LSLI are trademarks of Livermore Software Laboratories, Inc. IBM is a trademark of International Business Machines Corporation.

Circle No. 15 on Inquiry Card

In Which We Summarize

by Jeffreys Copeland and Haemer

With this column, we end our 17-part series on POSIX programming. Whew.

We called our series "POSIX from the Outside In" because we took an unusual approach: Instead of a methodical recitation of the interfaces, we asked, "What interfaces need to be in POSIX.1 to support the commands, like `ls`?" Face it: Standards aren't so exciting that just reading about them will make them stick in your head. Our goal was to spur you to learn the POSIX.1 interfaces by making you think about why POSIX.1 has the interfaces it does. (We also thought it would be a fun approach to try.)

Likewise, we tried to throw a few questions into each column to be answered the following month. This tune-in-next-week ploy wasn't just a cheap trick to entice you into reading our next column. It was another attempt to tease you into thinking enough about what you were reading to internalize the material.

We'd like to hear if our little techniques worked. Drop us a line and let us know–our email addresses are below. If you do send us feedback, we'll send you the punch line to "What's Dan Quayle's favorite palindrome?"

While we wait, here's POSIX.1's annotated table of contents.

Where We've Been

1. **From the Outside In** (August 1993). A brief introduction to the world of standards itself, a general cook's tour of the structure of the POSIX.1 standard and a sketch of how POSIX.1 relates to other standards you need to worry about.

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *lives in Austin, TX, where he consults, writes and raises children, cats and roses. His recent adventures include automating a series of landfills and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization and typesetting.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

2. **Headers, Identifiers and Writing Programs** (September 1993). Following a tour of the header files and the rules, POSIX sets out to keep you from stepping on the identifiers that are defined in those headers. We write simplified versions of two commands, `touch` and `ls`, in order to illustrate the way C programs use POSIX.1 calls to talk to the underlying operating system, and what those calls must provide in order to supply the functionality that we use day-to-day at the shell level.

3. **In Which We Write `ln`** (October 1993). Elaborating on the previous column's discussion of the `stat` structure, we discuss the underlying UNIX inode that `stat` was designed to report on, and use the `ln` command to introduce the idea of access functions that change the attributes reported by `stat`.

Most of the POSIX.1 standard deals with two things: file systems and processes.

4. **In Which We Move and Remove Files** (November 1993). On traditional UNIX systems, the commands `ln`, `mv` and `rm` are all the same program. The fourth column uses that fact to firm up the idea of links and inodes, to delve deeper into how to write commands and to provide readers with a much more concrete picture of what UNIX directories and filenames are really all about.

5. **In Which We Explore Directories** (December 1993). On traditional UNIX systems, directories are files. (On non-BSD systems, they're particularly simple, two-column files that pair file names with inode numbers.) We see here how POSIX frees itself from UNIX by turning directories into abstract data structures, with access functions that let you write truly portable versions of `ls`. We then elaborate on access functions by talking about the functions that underlie the familiar commands `chmod` and `chown`.

6. **In Which We Discuss File Attributes** (January 1994). Here, we round out our treatment of the `stat` structure by looking at file types other than regular files and directories, and the three "times" found in the `stat` structure: `st_ctime`, `st_mtime` and `st_atime`. We reinforce the idea that POSIX.1 is full of access functions that report on or change underlying operating system data structures. In the process, we write more sophisticated versions of both `ls` and `touch`, shell-level commands that report on and change the fundamental file-system data structure, the inode and touch on `mkfifo`.

7. **In Which We Discuss File Contents (Finally)** (February 1994). Having finally nearly exhausted the inode, we use one last field, `st_size`, to delve into both the file itself and the access functions needed to change file contents. Our command? `cat`. And once we get started writing it, we don't stop until we have you hip-deep in the question "What POSIX interfaces underly the familiar standard I/O library?" and its corollary, "Why don't we use them instead?"

8. **In Which We Discover Processes** (March 1994). Most of the POSIX.1 standard deals with two things: file systems and processes. This isn't a limitation on POSIX.1–it's mostly what operating systems deal with. In this column, we quickly review what POSIX.1 says about file systems and move on to processes. We introduce the two most important process-related interfaces: `fork()`, which creates new processes, and the `exec()` family, which ties processes to files. To stress the need for these interfaces and to help clarify why they're different, we look at logging in in some detail, concentrating on the `getty`, `login` and `passwd` commands.

9. **In Which We Corral Some Processes** (April 1994). We've mentioned that the two central concerns of operating systems are file systems and processes. On POSIX systems, both form trees. In this column, we explore the process tree and provide a shell script to display it on your system. The script also gives us an opportunity to touch on POSIX.2–the standard that provides for portable shell scripts.

10. **In Which We Look at the Processes in the Corral** (May 1994). We continue exploring our rough analogy between processes and files to see how far it will get us. This also leads us to ask what the attributes of a process might be and how we can go about getting them. It turns out there isn't a system call analogous to `stat()` that gets all the process attributes in a single call. There is, however, a shell-level command (which, unfortunately, isn't standardized by POSIX.2), that gets the information: `ps`. We thus spend a little time talking about `ps`.

11. **In Which We Go to the Beach** (June 1994). No series on system calls would be complete without a column on how to write a shell. Starting with the now-familiar `fork()` and `exec()` calls, we lead onward to talk about the underlying calls needed to implement I/O redirection.

We also dip our feet into the sea of system calls provided to get process attributes by implementing a simple version of time.

12. **In Which We Check Out the Clock Factory** (July 1994). In this column, we let ourselves get diverted into contrasting the three different time-related headers specified by POSIX.1: `<sys/times.h>`, which deals with timing processes, `<utime.h>`, which handles the times in the `stat` structure and `<time.h>`, which is concerned with wall-clock times. (We don't have much of an excuse for including this distraction, except that we don't have an excuse

SALES • RENTALS
CONVERSIONS • INTEGRATIONS

RS/6000
RT/6150
SERIES/1
SYSTEM/36
AS/400

IBM
Authorized
Distributor Product
Integrator

CALL:
(800) 888-2000

Dempsey
BUSINESS SYSTEMS
Where IBM Quality Is Second Nature.

18377 Beach Blvd., Suite 323
Huntington Beach, California 92648
(714) 847-8486 • FAX: (714) 847-3149

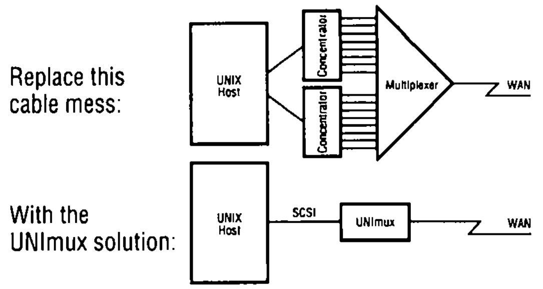
IBM is a registered trademark of International Business Machines Corporation.

Circle No. 9 on Inquiry Card

NEW FROM DCB

CLEAN UP CABLE CLUTTER –AND REDUCE YOUR COSTS– WITH THE NEW DCB UNIMUX

Eliminate that snarl of wires to your standard multiplexer. With the DCB UNImux, you can make ***all*** your local connections with a single cable to a SCSI interface and reduce the number of slots used.



For AIX, HP-UX, SCO, DEC and SUN.

Data Comm
for Business, Inc.

Manufacturers of modems, multiplexers, everything for your data communications application.

807 Pioneer, Champaign, IL 61820

Toll-free: (800) 637-1127

Illinois: (217) 352-3207

Circle No. 7 on Inquiry Card

for putting it anyplace else either, and we do need to talk about it.) The discussion of `<utime.h>` also allows us to touch on internationalization, a fascinating subject.

13. **In Which We Discover Signals: The Other IPC** (August 1994). We return to processes and consider the problem of moving information in and out of them. Having written about pipes when we write about I/O redirection, we now attack the issue of the only other interprocess communications mechanism specified in POSIX.1: signals. Faced with two fundamentally incompatible ways of handling signals (AT&T- and Berkeley-style), POSIX sidestepped choosing one or the other and invented a suite of interfaces for reliable signal handling that's basically sane but requires rewriting your code. This column goes through what you need to know to rewrite, culminating with `sigsetjmp()` and `siglongjmp()`, which give you the option of preserving the signal mask during nonlocal go-tos.

14. **In Which We Discuss the Environment** (September 1994). We've already admitted that there is no specific system call to get all the process attributes. Nevertheless, at this point, you may have concluded that there must be system calls to support commands like `logname` and `ps`, and shell functionality such as environment variables. In this column, we summarize those functions.

Looking back at this column, we realize with embarrassment that we failed to credit Fred Zlotnick's outstanding book, *The POSIX.1 Standard: A Programmer's Guide*, for the table of process attributes. This book, Richard Stevens' *Advanced Programming in the UNIX Environment* and Marc Rochkind's *Advanced UNIX Programming* are musts for the serious UNIX programmer.

Generalizing the topic of "environment" just a little, we also talk about the `sysconf()` and `pathconf()` functions, which POSIX gives you to let applications ask the system what optional behaviors it supports.

Finally, we return in this column to talk about calendars, more or less because we feel like it and it is, after all, our column.

15. **In Which We Discuss Replacements for the `ioctl()` System Call** (October 1994). There are only half a dozen classic UNIX systems calls for dealing with files: `open()`, `close()`, `read()`, `write()`, `seek()` and `ioctl()`. The reason there are only half a dozen, despite the fact that even devices are files, is hidden in that last call, `ioctl()`, which is a grab bag of device-specific functionality. This makes `ioctl()` impossible to standardize; POSIX.1 has no standard for `ioctl()`. That said, a handful of device-specific actions–those that provide terminal control–are so frequent that no operating system can be without them. These must be standardized to guarantee portability of a wide variety of key, "raw-mode" applications, such as screen editors, that need direct access to keystrokes and screens. In this col-

umn, we discuss the new POSIX.1 interfaces that provide that functionality.

16. **Odds 'n Ends** (November 1994). POSIX.1 was designed as an interface standard, not a protocol standard; its goal is to make your source code portable, not your data. If POSIX.1 stuck strictly to this goal, however, we programmers would be left in a paradoxical position: We'd have portable code, but no way to move the programs from one machine to another. To solve this problem, the standard sets out two formats, standardized versions of `tar` and `cpio`, that all conforming systems must support.

In addition to `tar` and `cpio`, we use this column to talk about odds and ends, like `fdopen()` and `fileno()`, which interconvert standard C's streams and POSIX.1's underlying file descriptors, and the new POSIX.1b standard, which specifies real-time extensions to POSIX.1, including scheduling, timers and a suite of additional interprocess communications mechanisms.

Highlighting the POSIX Standard

What would you say if your boss were to ask you for a single-sheet, management summary of this series? Here's our answer.

POSIX is a suite of international standards for programming interfaces to any operating system. Of these, POSIX.1, the system interface standard, is now everywhere. Conform to it, and your code will be portable.

POSIX.1 is modeled on traditional UNIX practices (particularly, but not exclusively, System V). Despite this, acceptance of POSIX is so widespread that many non-UNIX systems are now certified as POSIX-conforming, including IBM's MVS, HP's MP/E, Unisys' CTOS and DEC's MVS.

POSIX.1 interfaces are sometimes more abstract than their traditional UNIX counterparts. This is to avoid tying the interfaces to UNIX or to specific hardware.

In a few cases, where folks thought a standard was necessary but there wasn't consensus, POSIX.1 created headers, symbolic constants or interfaces to supply the needed functionality.

Where behavior is optional or implementation-dependent, POSIX.1 provides the means for the application to determine an implementation's choices.

Hasta La Vista, Buckaroos

And, as the sun sinks slowly in the West, we steal a page from Roy and Dale, and wish you happy trails. If you've been with us for this whole series, then we hope we've given you what you needed. (If you've been with us since our earlier series on internationalization, then you've probably gained the answer to another *RS/Magazine* trivia-quiz question.) If you've just joined us, then we hope to see you again. ▲

RS/6000™ users... Cambex is the only choice.

Fast Delivery

- New HD1 & HD3 Memory Boards & SIMMs 0 - 256MB
- SCSI disk subsystems 1 - 9GB Internal/External FWD
- AIX software utilities Back-up/Restore AIX Mirroring Tape Duplication
- Tape back-up 20 -100GB, 9GB/HR Internal/External/Autoloader
- RAID disk subsystems 4 - 189GB Fully Redundant
- Cambex products are eligible for IBM maintenance

Cambex is the **ONLY** alternative manufacturer of new HD1 and HD3 compatible memory boards for your RS/6000™ system.

CALL TODAY for a quick price quote from our Cambex***Direct*** representatives.

VISA MasterCard

1-800-292-RISC

RS/6000 is a registered trademark of the International Business Machines Corporation

Circle No. 3 on Inquiry Card

New for the RS/6000!

Tritus SPF™ and IBM REXX/6000™

The editor and macro language you already know are now available for AIX!

Reliability. Tritus SPF has been the SPF editor of choice for over two years in the DOS and OS/2 world. Now the power of mainframe editing is available for AIX with all the same functionality and ease of use.

Compatibility. Tritus SPF emulates ISPF/PDF including modifiable panels and keyboards. Plus, it supports many other features like Undo/Redo and highlighting.

REXX Macros. Write your own edit macros utilizing REXX and an ISREDIT superset of over 95 commands.

Great Deal. Tritus SPF and REXX/6000 are available as a package, or they can be ordered separately. Call us at **1-800-321-2100.**

TRITUS

6034 W. Courtyard Drive, Suite 120
Austin, Texas 78730-5014 USA
(512) 794-5800, Fax (512) 794-3833

Circle No. 17 on Inquiry Card