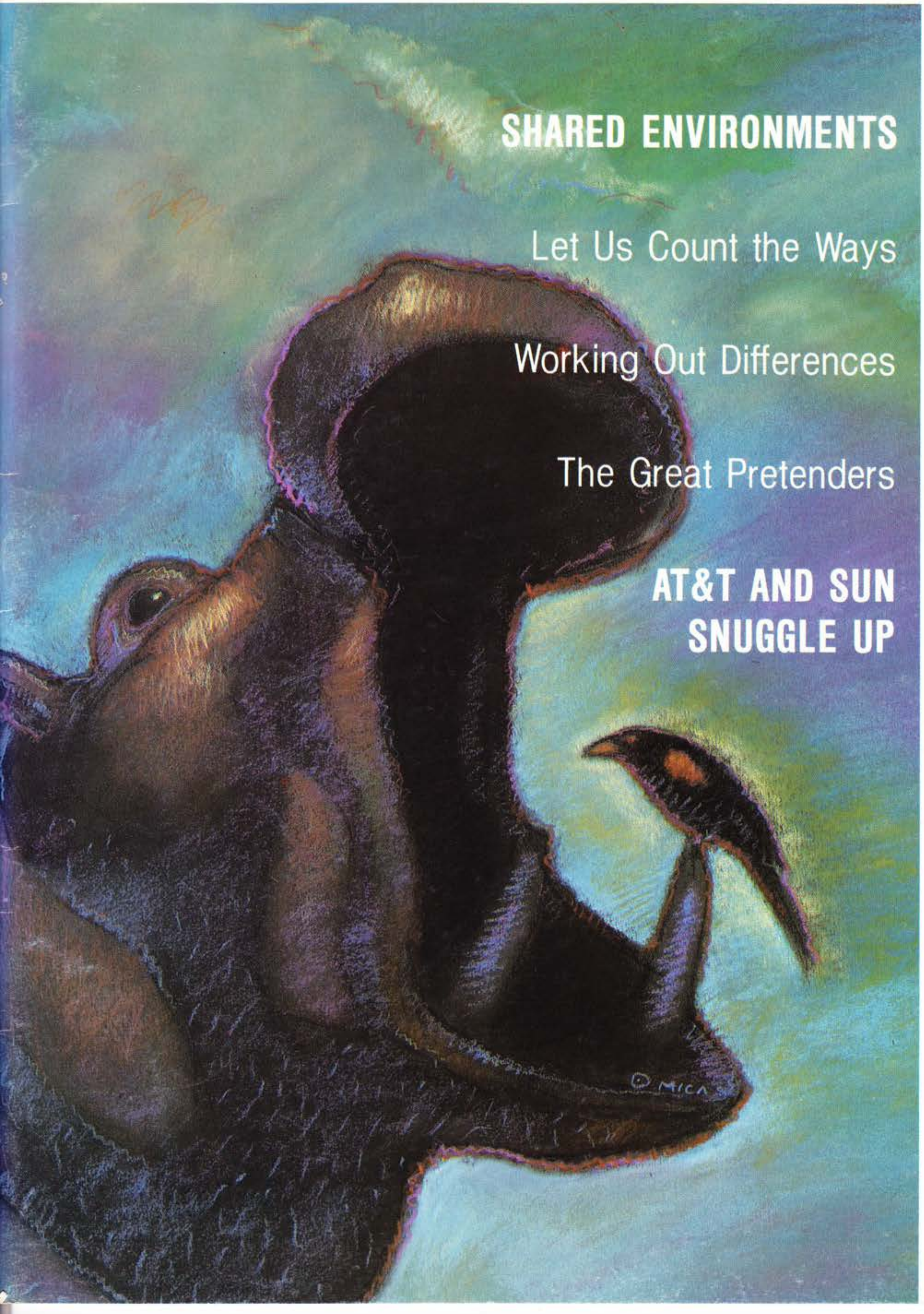
UNIX® REVIEW

THE PUBLICATION FOR THE UNIX COMMUNITY VOLUME 6 NUMBER 1

$3.95



SHARED ENVIRONMENTS

Let Us Count the Ways

Working Out Differences

The Great Pretenders

AT&T AND SUN SNUGGLE UP

Astonishment. Any computing environment that presents UNIX interfaces as a subset of some larger system needs to preserve as many of the original product's features as possible. In this context, of course, the exact definition of *original product* and *features* is problematic, but it is nonetheless safe to assume that no UNIX implementation will succeed unless it affords portability of programs and users, and unless it performs at least as well as a native kernel running on comparable hardware (especially when executing often-used primitives such as **stat** and **fork**).

Program portability is probably the easiest of these objectives to achieve, since most well-written UNIX programs require only that the operating system provide interfaces that have the correct semantics, and that the file system bear at least a vague resemblance to the typical UNIX file system (it's pretty hard to live without **/usr/tmp**). Even the less notable examples of good programming practice can usually be tricked into running in a non-native environment through the emulation of implementation-specific bugs and other undocumented features. But users—especially programmers and system administrators—are more sensitive. They come to your system with a variety of expectations and, of course, prejudices. Most users are not especially comforted by the idea that your version of UNIX adheres to the letter of an interface specification unless it also has the look and feel they're used to.

UNIX has always been characterized by a high degree of *conceptual integrity*, probably because it was developed and shepherded through its early years by a small group of like-minded individuals [3,4]. A system that has conceptual integrity is one in which fundamental concepts are manifested in similar ways at all levels. In such a system, anyone who has mastered the syntax of a few commands or system calls will find that the same rules apply, more or less, across the board. As a result, it is soon easy to infer the behavior of something you've never tried from the behavior of something you have tried, and the whole business of using the system becomes an order of magnitude easier.

Successfully extending a system that is lucky enough to have conceptual integrity requires that all extensions be tested against what has been called the Law of Least Astonishment. In this context, that means the behavior of additions to a system must be consistent with behavior patterns the system's users have come to expect from it. Extensions to UNIX that violate this law are less likely to be successful than those that don't.

Living with history. In 1979, there was a fair amount of debate among the architects of Apollo's kernel over whether it made sense to try to fit the features they wanted into the (Version 7) UNIXes available at the time, or whether they should just start over from scratch. And while the decision then was to start from scratch, the system that resulted bore at least a surface resemblance to UNIX, supporting a similar file system, stream I/O interface, and process abstraction. It also included the UNIX-like Software Tools shell and command set. But beneath it all was a new network-based kernel (written mostly in Pascal) [5] that supported features, such as large virtual address spaces and a large-scale distributed file system, that are only now becoming available on more traditional UNIX kernels. Initially, these features proved valuable enough to stand on their own, attracting third-party application developers for whom UNIX was not, at the time, an issue. Within a few years, however, the success of UNIX as a development and runtime environment for workstation software put increasing pressure on Apollo to find a way of letting UNIX programs run on Apollo systems. While Apollo's UNIX-like features may have given some comfort to UNIX users migrating to Apollo, they were no help to UNIX programs.

Early attempts to implement a hybrid 3BSD/System III UNIX solely in global libraries [6] enjoyed enough success to allow the porting of several hundred UNIX applications to Apollo systems. Subsequent revisions of the system

Not Just a Poor Man's UNIX

Providing a common UNIX-style interface across computing environments has made both users and programs portable.

BY D. SCHERRER

Early in 1976, before UNIX was available outside the university environment, Brian Kernighan and P.J. Plauger published an unassuming little book, *Software Tools*, in which they described the simplicity and elegance of the UNIX world view through example. Theirs was yet another experiment at bringing the UNIX environment into peaceful coexistence with a host system, and the examples they chose were portable implementations of some of the basic UNIX utilities. In effect, their book offered readers the means, using the UNIX model, to improve their computing environment even if all they had to work with was a hardware dinosaur running a minimally functional operating system.

Networked Computing. Researchers at Lawrence Berkeley Laboratory (LBL) quickly picked up on this opportunity to attempt to solve an immediate problem: how to make effective use of the dozens of operating system environments rapidly becoming accessible with the emergence of computing networks? By extending the abstractions and the tool set offered by Kernighan and Plauger, LBL's researchers were able to develop, in portable (and unlicensed) form, a UNIX interface that could be implemented on top of virtually any operating system*.

Because the researchers did not have the luxury of insisting that each piece of hardware on a given network run the same operating system, they designed their Software Tools Virtual Operating System (VOS) to provide both programming and user-level interfaces to the local system, however unusual or arcane. The package included a shell, 60 or so of the favorite UNIX utilities, and an extensive programming library.

To make the package portable, the researchers defined a VOS interface expected by the Tools source code. This interface included such operating system capabilities as file access, process control, directory manipulation, and command line argument handling, and needed to be implemented for each host operating system. Sometimes the mappings were straightforward; at other times they were complex. However, once the VOS layer was implemented, the source code for the Tools themselves compiled and ran essentially unchanged.

Guaranteeing Portability. Obtaining this amazing level of portability was, of course, the trick. The abstractions offered by Kernighan and Plauger were a starting point. But as more and more groups began to move the Tools to all manner of new systems, the real limits of portability were discovered. At that point, the VOS community came together, pooled its knowledge, energy, and experience, and eventually was able to specify a carefully chosen abstraction for the VOS interface. The interface was effective enough to allow the package to be implemented on more than 50 different operating systems.

Two "prime directives" guided the choice of the VOS interface:

- It could require no changes in the underlying operating system. This was an administrative necessity, if nothing else.
- The VOS tool set and library had to work cooperatively and in conjunction with the local system. Tools had to transparently support local file types and the local character set, although all visible VOS file and I/O handling was to be based on the UNIX model and the ASCII character set. Magic, newly-devised file types were to be avoided. The VOS shell had to be able at least to execute local utilities (if not always be able to use them in pipelines).

> *The Software Tools VOS Project grew out of a recognition of the need to provide for multiple operating system environments without sacrificing the advantages of a consistent and well-known interface. In the process, much was learned from analyzing what UNIX did well and what it did badly.*

While these requirements greatly increased the challenge faced by the VOS implementors, the result was a carefully chosen VOS interface layer and a user interface which effectively complemented, and cooperated with, the local system.

Portable Users. The VOS project addressed portability, then, from two directions. The original goal, at least as envisioned at LBL, had been to provide a common UNIX-style interface across computing environments, which would minimize the need to reimplement programs and retrain users—thus making people and programs portable, if you will. The method used to achieve this goal involved developing a UNIX-like package that would be readily portable to any operating environment.

At the time, unfortunately, neither the concept of portability nor the desire for a UNIX-like interface was enthusiastically embraced by manufacturers, who were busy basing their sales campaigns on their ability to provide something different and, therefore, potentially "better". Software portability and—to an even greater extent—human portability were viewed by those trying to sell systems as an annoyance at best. Only recently have such concepts as software portability and the desirability of providing a consistent interface become recognized as socially and economically valuable.

Poor Man's UNIX? The Software Tools VOS Project was one of the first attempts to provide a hybridized UNIX/local-host interface. To this day, many people think it was little more than attempt to build a "poor man's UNIX", or to provide a substitute for the real thing. Though it certainly played that role for many, the VOS project offered much more. It grew out of a recognition of the need to provide for multiple operating system environments without sacrificing the advantages of a consistent and well-known interface. The VOS flavor of UNIX was designed to cooperate with and complement the host environment, and not as a mutually-exclusive alternative. In addition, the Software Tools people learned many lessons from analyzing what UNIX did well and what it did badly, and in the process wrote many new chapters on portability, defensive programming, software design, even software management. ●

Deborah Scherrer is a computer scientist at mt Xinu. She previously did research at Lawrence Berkeley Laboratories, where she helped to found the Software Tools movement.

---

* Hall, Scherrer, and Sventek, "A Virtual Operating System", *Communications of the ACM*, Vol. 23, No. 9 (Sept. 1980).