ϕnix: A Unix Emulator for VAX/VMS

by

Michael Caplinger

August 10, 1987

Rice MASC TR82-8
φnix: A Unix Emulator for VAX/VMS

Michael Caplinger
Department of Mathematical Sciences
Rice University
Houston TX 77251

Abstract

Software performance and portability issues, (particularly the poor performance of the Unix Fortran compiler) have forced many VAX sites to reluctantly choose VMS over Unix\(^1\). However, the programming environment provided by VMS is much less convenient and user-oriented than the Unix environment, as well as being unfamiliar and restrictive to someone familiar with Unix.

One solution to the problem, the "Virtual Operating System" (VOS) of Lawrence Berkeley Labs, provides a Unix-like interface on top of an already existing operating system. It does not, however, address the problem of porting Unix programs. Using a different approach, David Kashtan of SRI wrote "Eunice", a Unix simulator. He attempts to solve the porting problem by emulating Unix at the system call level. Eunice has several problems. Some of the system calls are implemented inefficiently, and some incorrectly. Also, many factors make large amounts of source-level modification necessary to run arbitrary Unix programs.

Faced with this problem, we at Rice University are in the process of designing and implementing a complete emulation of Unix, called φnix, consisting of a set of system calls and a new loader developed for VMS. The system calls provide exact simulations of the facilities provided by the Unix kernel, even providing functions that are not provided by VMS. The loader takes Unix object modules and produces a VMS executable. This gives φnix complete object-level compatibility with Unix.

The development of φnix is being done in a somewhat-enhanced version of Eunice, which already uses the new loader. The completion of this project should make porting of nearly all Unix programs to VMS a trivial exercise, and will extend the benefits of the Unix community to VMS sites which, up to now, have been denied access to them.

\(^1\)Unix is a trademark of Bell Laboratories. VAX and VMS are trademarks of Digital Equipment Corporation.
1. History

In 1980, Rice University purchased a Digital VAX-11/780, and became embroiled in a controversy over which operating system to run on it. The user community was an unusually heterogeneous group. About one third of the potential users were computer scientists, who were used to the Unix environment and made heavy use of available Unix tools. The second third were numerical analysts, whose prime concern was the development of Fortran programs. The remaining users were biochemists working on large, CPU-intensive molecular models. The needs of two thirds of the user community, then, could not be served by Unix, since Unix does not support a optimizing Fortran compiler and a fast run-time system for Fortran. Also, the biochemists were interested in using already-developed programs, nearly all of which had been written under VMS. On the other hand, the computer science group was unwilling to abandon the tools and large program base offered by Unix. The dilemma appeared insoluble, but the larger group prevailed and VMS began running in the fall of 1980.

Some hope was offered by the availability of Unix emulators: systems that run under VMS but provide an environment similar to that of Unix. There were only two available: the IS/1 Programmer’s Workbench system from Interactive Systems Corporation, and Eunice, written by David Kashtan at SRI [1]. Aside from technical considerations, ISC’s product was too expensive for Rice to afford, so Eunice was chosen by default.

After Eunice arrived and was installed, it became obvious that it was a useful tool, but far from ideal. Attempts to bring programs over from Unix met with frustration, and code often had to be heavily modified in order to work. Also, the environment provided by Eunice had several annoying differences from the familiar Unix world.

In the summer of 1981, after establishing a barely usable environment at the cost of a large amount of effort, a project began to enhance the performance of Eunice. This project later evolved into a full-blown effort to produce a new system, which was to be a complete emulation of Unix.

2. Eunice

As delivered from SRI, Eunice consisted of three elements:

(1) A set of VMS executables of most of the common Unix utilities;
(2) A VMS object library containing most of the Unix system calls;
(3) A modified Unix assembler, that could take programs in Unix assembly language and produce VMS object modules, that were then linked, by the VMS linker, into a VMS executable.

This system was sufficient for compiling relatively simple programs, but had some problems. The fact that Eunice relied on the VMS linker was the source of much difficulty, as it made no distinction between upper and lower case. All Unix code reliant on the uniqueness of external names with the same characters but different case mixtures (e.g., Xflag and xflag) had to be modified. Also, the common Unix practice of including header files with external variable definitions at the top of each module in a program led to problems, as the VMS linker considered such variables to be multiply defined.
Another major difficulty was caused by the file names available under VMS. While Unix names are 14 characters long, with almost all characters allowed, VMS names have a 9 character body and a 3 character extender. The characters must be alphanumeric, and there is no distinction made between cases. Many source modifications had to be made to keep from generating illegal VMS names, and makefiles had to be rewritten, as the file names they referred to had been changed.

But the most serious problem arose because Eunice was not really designed to function as a complete Unix environment. Unix utilities could be run from the VMS command interpreter, but because of synchronization problems, commands run under the Unix shell often failed. Many of the system calls did not function exactly as they would under Unix, and others were unreliable.

The first task undertaken at Rice was the construction of a new loading scheme.

3. The Rice Loader

Since the VMS linker was the source of much confusion, it was decided to remove it from the compilation system entirely. The standard Unix loader ld was used as a base. It took a series of Unix format object files and produced a single Unix object file, whose only unresolved references were to Unix system calls provided by Eunice. At this point, a second phase of loading was invoked. This phase could have read VMS format object libraries and linked in their code. But VMS object modules have a very complicated format, and another option existed. VMS supports an facility called a shareable image: a segment of position-independent code that is only mapped into the user’s address space on execution, and whose text is shared among all processes using that code. Not only does this increase the efficiency of user programs (and reduce their size on disk), but the shareable image can be dynamically changed without relinking all programs that use it. Fortunately, Kashtan had provided the Eunice system calls in this form as well, and the Rice loader linked against this Eunice shareable image [2].

This solved many problems immediately, and a large number of Unix utilities were brought up immediately, such as Unix Emacs, vi, the Berkeley 4.1 C compiler, the Berkeley Pascal compiler, the Cornell editor ged, and the Cornell shell editor. Also, since this scheme gave us complete compatibility with Unix object files, we were able to begin using new Unix math, I/O, and database libraries. The loader proved so successful that we distributed it to several sites, and Kashtan began including it with Eunice as well.

It soon became obvious, however, that even though the Rice loader had made porting programs from Unix much simpler, it could not solve other fundamental problems with Eunice. Process creation calls were particularly buggy, and the fork system call was not provided at all. Only the Berkeley vfork call was available, so many utilities, including the Bourne shell, would not run without a large amount of modification. Problems also existed in status codes passed between processes by exit and wait, in the signal system, and in handling of file descriptor inheritance.
4. \( \phi \text{nix} \)

There are several possible strategies to provide the functionality of two operating systems on the same machine. One possibility is to design an operating system kernel, and write all higher levels using its primitives. Unfortunately, it was a bit late in the day to reimplement VMS.

A second possibility is to run the two systems in a parallel fashion. For example, Unix could be run as a single process under VMS, and the two systems could each have devices controlled only by it. However, this scheme precludes simple communication between users of "virtual Unix" and VMS.

A third approach, used in Eunice and adopted by us as well, is to provide the primitives of one operating system in terms of the other. This has the virtue of being simpler than the other approaches, as the host operating system (VMS, in our case) already provides a higher-level view of the machine.

In the fall of 1981, Kenneth Zadeck began designing an emulation of the Unix I/O calls for VMS. After two months, it was determined that a project would be undertaken that would produce as close a Unix emulation as was possible under VMS. This project was dubbed \( \phi \text{nix}. \)

Zadeck continued on the I/O system, and was joined by David Johnson and Paul Milazzo, working on the process management calls, and Michael Caplinger, who wrote the signal handlers and served as project librarian. Scott Comer provided his knowledge of VMS internals. The project was directed by Professors Corky Cartwright and Robert Hood.

\( \phi \text{nix} \) was certainly inspired by Eunice, and was developed under the Eunice environment, but its goal was a more ambitious one. Rather than providing a system for "porting Unix programs," \( \phi \text{nix} \) was to be a complete simulation of Unix, making it possible to run nearly every program without modification under VMS, and to provide a user interface nearly indistinguishable from Unix. At the same time, however, it was to be compatible with VMS, so that \( \phi \text{nix} \) and VMS users could easily interact, and that VMS users could use Unix utilities without invoking the Unix shell.

4.1. Implementation

VMS is a large virtual memory operating system derived from the earlier RT-11 and RSX-11 operating systems from DEC [3]. It provides facilities for process creation and intercommunication, a tree-structured file system, and a large number of file formats. Unfortunately, the primitives of VMS are not well matched to the requirements of a Unix emulator.

4.1.1. Process management

Unix provides two primitives for process creation: \texttt{fork}, that creates a new process with the same address space contents as the creator, and \texttt{exec}, that overlays an existing process with a new image. VMS provides only one mechanism, called \texttt{CREPRC}, that creates a new process running a specified image. It is therefore necessary in \( \phi \text{nix} \) for \texttt{fork} to first create a new process running the same image as the creator, but in a virgin, inactive state, and then to copy user data and other context information into the new process before allowing it to run.

---

\(^1\)pronounced "Phoenix".
**exec** must interface into the VMS image activator, which is not a defined interface in VMS, in order to overlay a new image. It must also save some of the old image's state during the overlay operation, so that both file and signal context and registers are maintained across **exec** boundaries. Extensive use of the VAX access mode structure is used in these calls.

### 4.1.2. I/O

Unix is well known for its simple, uniform interface to input/output devices. VMS is a much more traditional approach, with a proliferation of file types and a baroque interface to the system I/O primitives, typically not used in applications programs.

VMS provides two interfaces to the I/O system. The higher-level interface, RMS (Record Management Service), is device-independent and provides buffering. The lower-level, QIO, provides different functions for each device type, and does not buffer. In spite of its greater complexity, *unix was forced to use the QIO interface, because only under it was enough control possible to implement all Unix functions. For example, file sharing among processes could not be provided under RMS. *unix can do this by implementing Unix-like buffering and file tables internally, and using QIO to perform raw I/O, in much the same way that Unix device drivers communicate with physical devices.

The primary file type supported by *unix is the Unix-format 512-byte fixed record file. The system calls produce such files by default. On input, VMS *textfiles* (variable length records, implied carriage return) can also be read, and appear to *unix programs as normal Unix files with imbedded newlines. This is important since VMS editors and compilers write, and expect to read, such files.

Other file formats are not deblocked, but are treated as "raw" files. Reads and writes on them operate on single records, rather than the requested number of bytes.

Since Unix programs often read directories, *unix translates VMS directory information into Unix format, in a manner transparent to the calling program. This allows such programs as **find** and **ls** to work without modification.

Unix pipes are implemented with the more general VMS "mailbox" facility. A mailbox is essentially a named virtual file, that can be accessed by anyone who knows the name and has file system permission. *unix associates a mailbox with each created pipe, and passes the name of the mailbox to the children of the creating process. The same startup module that copies the child context after a **fork** opens any such mailboxes, completing the pipe connection.

Terminal management in *unix is done internally, with no processing by the VMS terminal driver. This is done so that Unix control codes (such as interrupt, delete, and end-of-file) can be reset, and so that Unix-style character display can be done.

### 4.1.3. Signal Handling

The Unix **signal** system provides a mechanism for detecting faults during execution, and also receiving messages from the user and from other processes. VMS has facilities for doing both, but they are not integrated. Faults and interrupts caused by memory exceptions, division by zero, and the like are caught by the VMS "exception vector" facility, while
asynchronous actions, such as a timer or keyboard interrupt, are processed by the "asynchronous system trap" (AST) mechanism. These are messy but fairly straightforward primitives.

A more difficult problem was posed by the Unix kill system call, that allows one process to cause an exception to be raised in another. Since ASTs can be generated by device drivers (for example, the terminal driver generates one when the interrupt key is pressed), a device driver was designed whose sole purpose was to deliver ASTs to target processes on request. This allowed kill to be simulated exactly.

4.1.4. File Names

A major difference between Unix and Eunice, apparent even to the casual user, is the range of legal file names. Since this discrepancy made many source-level modifications necessary, it was resolved to make this as minor a problem in φnix as possible. But since compatibility between φnix and VMS users was a prime consideration, internal table schemes were rejected. The current scheme maps 14 character Unix names into legal VMS names by removing characters when necessary. However, many idiosyncratic Unix names (such as .profile, .rofix, and Makefile) are mapped into special VMS names (0PROFILE, 8ROFIX, and 9MAKEFILE). By keeping the conversion rules simple, VMS users can recognize the names when encountering them. Even so, we are still working on alternate solutions, as naming still constitutes a major point of incompatibility.

4.2. Remaining Problems

It would be nice to say that φnix is a perfect emulation of Unix, and that every program that runs under Unix is available to the φnix user without change. But some problems still remain. First and foremost, since the VMS executable format is different from that of Unix, any program that explicitly uses the Unix format cannot be ported. This makes support of the Unix symbolic debugger sdb impossible. (Of course, the VMS debugger is provided, and has many of the same features.) Features of other programs, such as the dump Lisp facility in Franz Lisp, also cannot be supported (although slight modifications in Lisp provided a similar mechanism).

Also, the necessity for name conversion requires that some programs be modified. The make utility is a good example. Since make uses names that come from the description file, and so never pass through the φnix conversion routines, calls to those routines must be inserted into the make source code.

5. Future Directions

φnix is currently about a month from completion¹, and will provide Rice users with a complete Unix environment. We are already contemplating other future projects. Although the initial φnix implementation was ad hoc, we have been examining the Bell and Berkeley Unix kernels, and believe that we can define a set of primitives that can be supported on any reasonable operating system. These primitives are at a lower level than the Unix system calls, and provide basic functions such as process creation, suspension, and contents replacement;

¹ as of 1 July 1982. This estimate proved to be wildly optimistic! See Appendix A for distribution information.
character and block raw I/O; and interprocess communication. We intend to put them to the test by reimplementing the phoenix system in a transportable fashion for VMS, and then moving it to a Data General MV/8000 running AOS. If this attempt is successful, it will be a major milestone in portable operating system design.

The current phoenix system will extend the benefits of membership in the Unix community to sites which have been forced to run VMS. Hopefully, in the future we will be able to provide these benefits to many other sites and operating systems as well.

6. References


Solaris will be in the public domain. We will make it available (for a nominal distribution charge) to all sites which have a valid Unix license from Bell (as we distribute a large amount of Unix source). Sites which have binary licenses only may be able to receive binaries; we are waiting to see what the demand for this is.

We currently hope that distribution will start in January 1983. Some sites, preferably those with VMS and Unix expertise, may be able to get testing versions earlier. If you are interested in getting a Solaris distribution, send information about your system configuration (such as CPU type, amount of memory, number and types of disks, amount of free disk storage available for Solaris, version of VMS you intend to run) to:

Myrna Turner
Dept of Math Sciences
Rice University
PO Box 1892
Houston, TX 77251