UCRL--102872

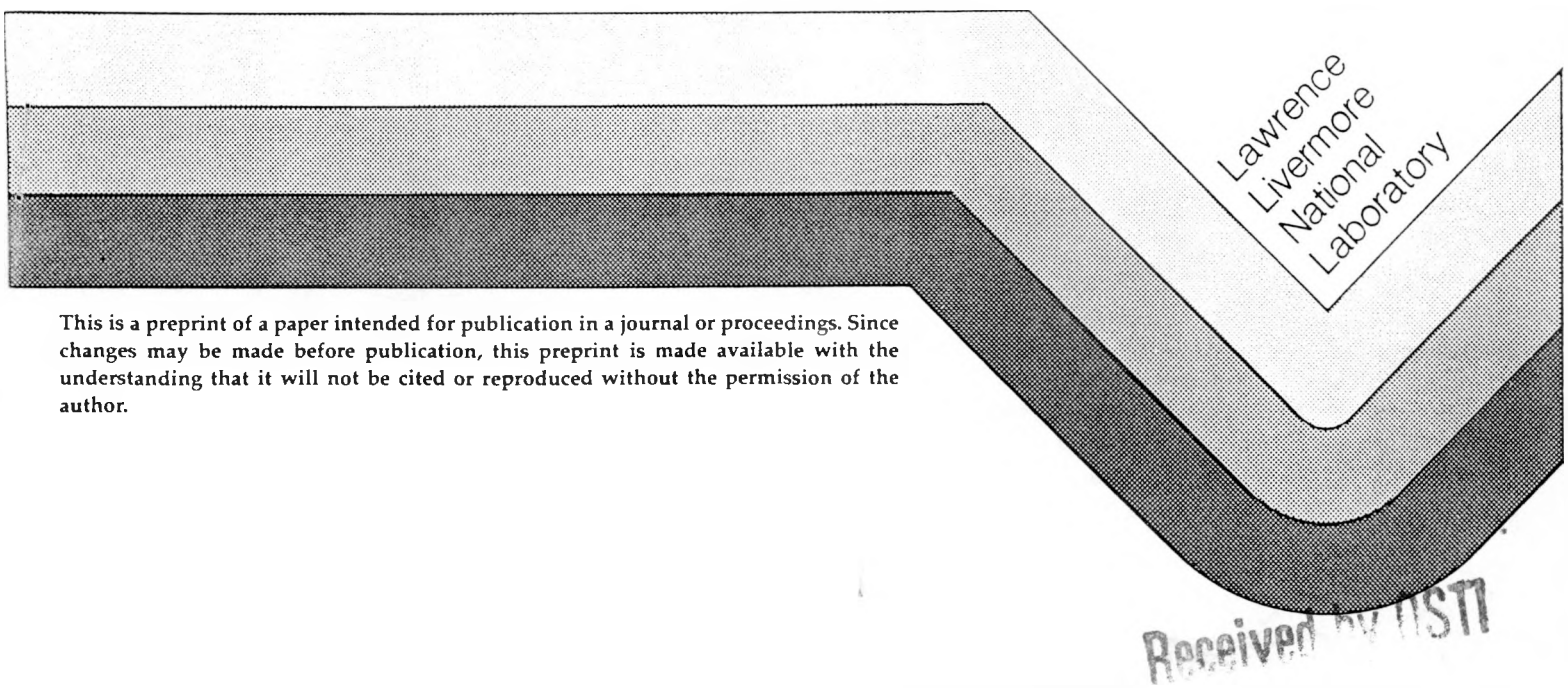
DE90 007651

Coping with Nonuniform Unix Interfaces in Designing a Portable Network Driver

Rich Wolski

This Paper Was Prepared For Submittal To
Usenix Conference Proceedings
Usenix, Anaheim CA

June 11-18, 1990



Coping with Nonuniform Unix Interfaces in Designing a Portable Network Driver

by

Rich Wolski
Computation Department
Lawrence Livermore National Laboratory
Livermore, CA.

This paper represents a draft of a proposed submittal for the 1990 Summer USENIX conference.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Abstract

The need for portable software continues to increase, especially within the Unix community, where a uniform interface is assumed to exist. Unfortunately, system software (particularly that which executes as part of the kernel) often has not been written to take full advantage of the homogeneity offered by Unix. Further, there is a need even within the kernel for greater uniformity of interfaces.

This paper describes some of the issues associated with the design and implementation of a kernel-level protocol driver which is portable among various Unix implementations. It also recommends areas where interfaces could be more uniform across implementations.

Introduction and Summary

The need for portable system software is most apparent given the nature of heterogeneous distributed computing. At the Lawrence Livermore National Laboratory (LLNL), we have designed and implemented a layered proprietary interprocess-communication (IPC) environment that is portable across multiple hardware and software platforms. Lightweight tasking is used to support multiple streams of execution within the IPC system, as well as to manage protocol state.

During the design phase of the project, we assumed that the Unix* device driver interface had been standardized across various vendor implementations to allow us to write a portable kernel-level driver for our proprietary transport through link-layer protocols. However, while much work has been done to standardize the system-call interface to Unix, we found that the interfaces to various kernel services (memory management, process scheduling, etc.) were widely different from vendor to vendor, and even from release to release of the same vendor's operating system.

This paper discusses the design and implementation issues associated with portable Unix drivers and some of the problems stemming from a lack of standard internal kernel interfaces. On the basis of this experience, we recommend that the following improvements be made within Unix to facilitate portable system software:

- Provide a uniform buffer-management interface across vendor implementations of Unix.
- Add a standard lightweight tasking facility to the Unix kernel.
- Devote a portion of the *proc* and *user* structures to maintaining state information on a driver-by-driver basis.
- At some level in the Unix kernel, provide a uniform device-access interface that is independent of any higher-level protocol.
- Provide a uniform set of signal semantics and an accompanying kernel-level interface.
- Standardize the interface of the timeout/untimeout facility across Unix systems, and provide an efficient mechanism to both set and cancel timers.
- Standardize the method of parameter passing between user space and system space.
- Include preprocessor functionality as part of the ANSI standard for the C language.
- Make a uniform clock interface with a standard resolution available from all kernels.

Background

To support distributed resource sharing in a heterogeneous environment consisting of computers ranging from PCs to supercomputers, LLNL's Livermore Computer Center decided to implement a distributed operating system to facilitate integrating these diverse systems [1], [2]. The LINCS (Livermore Integrated Network Computing System) software architecture defines a set of standard abstraction at the operating-system level (e.g., file, process, directory, clock, account, etc.), each managed by its own service. Servers manipulate the various implementations of these abstractions on behalf of clients who communicate their requests via a locally developed IPC model and set of communication protocols (application layer through link layer) [3], [4], [5].

For efficiency, LINCS has been implemented as the native operating system for the Cray X-MP and Y-MP hardware platforms under the name NLTSS (Network Livermore Timesharing System). It was clear that we did not have the manpower to support locally developed operating systems for every piece of hardware that anyone might wish to integrate into the system. Therefore, the architecture was implemented as a guest layer on other platforms and operating systems. Because LINCS requires only a uniform IPC interface, it did not make economic sense to reimplement the majority of the other operating system components (device drivers, trap and fault control, bootstrap programs, etc.) for each new piece of hardware to be added to the system.

Fortunately, the Unix operating system emerged as a growing standard in the scientific computing community and was available for most of the support machines that we planned to integrate as servers into the system and that users would use as workstations. Since the Unix system-call interface is mostly consistent across vendor implementations, application codes written in C should be portable, without much difficulty, between arbitrary network nodes. Furthermore, the Unix device-driver interface (which was also becoming standardized) provided a convenient way for the LINCS IPC facility to be made portable as well. However, we found that a lack of standard interfaces to internal kernel subsystems made producing such a driver more difficult than we had originally expected.

General Requirements

The primary requirement for our driver was portability. Not only did it have to run as a standard driver in a variety of Unix "flavors", such as 4.X BSD, DEC Ultrix, SunOS 3.X and 4.X, and Amdahl UTS, but it also had to be easily convertible to run under other operating system configurations. For example, the design was not to preclude a version for VMS or MS-DOS and therefore should modularize those sections of code needed to support operating system dependencies. Our driver was initially targeted for implementation under the four architectures listed above, each of which had different performance requirements and memory space limitations. Further, we knew we would have to port the system across many upgrades of a given vendor's hardware and

operating systems. This set of requirements contains conflict. One example is the classic trade-off between execution speed and memory usage. In cases where time efficiency strongly conflicted with space efficiency, we usually compromised in favor of the former. Another, often less well recognized, conflict is that which may exist between efficiency and portability. To get the driver running without Unix source code changes in so many environments required that we use a minimum set of universal system functions. If we were to make greater use of individual architectural features, we could improve overall performance at the expense of source code portability.

Use of Lightweight Tasking

We felt that it was important to structure our network driver around a tasking paradigm (as opposed to using the traditional Unix interrupt service mechanisms) to improve the extensibility of our implementation. Experience with research systems like Thoth, [6], Tunis [7], and V [8] has shown that lightweight tasking greatly improves modularity, since computations can be encapsulated with their stack state [9]. Thus, it is not necessary to maintain a series of state records which must be parsed each time an event occurs, as long as the correct task for that event can be scheduled—any needed state is simply embedded in the task's stack.

In addition, since tasks constitute logically parallel, independent threads of execution, their use seems to naturally fit the similar nature of protocol processing (i.e., activity on one protocol stream is logically independent of that on another, unless the protocol explicitly says otherwise).

Tasks retain state over periods of both execution and dormancy; this can be used to store communication state. The task scheduler must be able to find the appropriate task state based on the incoming information, but no explicit search is required for a communication state variable of any kind. Protocol modules are more easily programmed as tasks, since they can be written to naturally use their inherent memory to store communication state with no extra search cost. Tasking also extends to handle the requirements for parallelism which accompany multiprocessing. Since streams within a given machine are independent, tasks handling those streams can run in parallel if the hardware supports concurrency. While it is true that task scheduling adds some overhead to the uniprocessor implementation of our driver, we believe that the ease of extensibility offsets the processing cost.

Unfortunately, Unix does not typically provide a lightweight tasking facility within the kernel or an interface which allows driver codes to manipulate execution stacks in order to implement their own tasking. The C-library functions *setjmp* and *longjmp* provide a way to switch from one stack context to another, but their semantics seem to be somewhat ill defined. Under various implementations of Unix, the user of these functions is assumed to be "rolling back" to a previous stack context, specifically, to recover from an error condition [10]. To implement lightweight tasking, however, the task scheduler must be able to switch between noncontiguous stacks arbitrarily, which violates the simpler error-recovery model. While we eventually did have some success using *setjmp* and *longjmp* (in some cases by simply rewriting them to do full stack-frame reads and writes), the lack of consistency in their implementation

across different brands of Unix caused us to use the kernel *timeout* mechanism to trigger our task scheduler. This gives us a uniform IPL (interrupt priority level) on systems with interruptible kernels so that we do not have to worry about tasks interrupting other tasks. We have also taken the liberty of organizing our task scheduler as a large monitor to manage the critical regions that exist where communication state memory is accessed. Combining the task implementation with critical region management simplified the protection of critical regions without significantly constraining the tasking model. Although mixing the concepts of synchronization and scheduling in the same implementation is not as general as possible, time constraints forced some compromises in our implementation.

Facilities We Could Not Use

There were several features of various Unix implementations that we wished to take advantage of, but could not because these features were not universally available, at the time, in the brands of Unix we were targeting. We therefore took steps, quite often at the expense of efficiency, to avoid these features in favor of portability.

Buffering. We had hoped to use the native buffer management routines for each system, which typically are implemented around a data structure called an *mbuf* [11]. Unfortunately, not all systems support mbufs for internal network buffer management. Among the ones that do, the mbuf routines and macros do not present a uniform interface to the driver code. Consequently, we decided to implement our own packet management routines, based on wired-down kernel memory which we manage with our own allocation/deallocation scheme. On systems which do support mbufs as the interface to a particular hardware driver, we were forced to convert between our own packet format and mbufs (and vice versa), which entails copying the data from one structure to another.

A uniform buffer management interface across vendor implementations of Unix would allow us to avoid the inefficiency of an extra data copy.

Tasking. Since the tasking model of concurrency fit our driver design well, we hoped to use some Unix facility (like *setjmp* and *longjmp*) to implement lightweight tasking within our driver. Unfortunately, the behavior of *setjmp* and *longjmp* is very much implementation specific, and not always suitable for task stack manipulation. We therefore decided to use the Unix *timeout* mechanism, which seemed to be available in most Unix kernels, to simulate tasking where ever possible. Systems with interruptible kernels usually offer some kind of efficient software-interrupt facility which can be used to trigger network drivers, but the interface to this facility, if it is offered at all, is not standardized.

Since tasking seems to be a useful facility for implementing protocol drivers, we recommend that a standard lightweight tasking facility be added to the Unix kernel. Barring that, a uniform method of stack manipulation should be provided to allow drivers to implement their own versions of lightweight tasking.

Process state. There are several instances where our protocol implementation must keep state on a per-process basis. The most natural method for managing this state would be to add it to the Unix *proc* and *user* structures (depending on whether it needs to be accessed context independently or not), but these structures do not typically provide the necessary "hooks" to add driver-related state. Therefore, we, added our own per process data structures which include pointers to the Unix versions. When such state is needed, the driver code will search its per-process records and compare or dereference these Unix data structure pointers.

It would be useful if some portion of the *proc* an *user* structures could be devoted to maintaining some information on a driver by driver basis.

Existing driver interfaces. On systems where an existing Ethernet driver was available, we hoped to use this software as the interface to the physical layer without modification. Fortunately, we thought, the "if" interface (which at least BSD-derived versions of Unix support [12]) would provide us with portable network access to the physical communication medium. In fact, not only does the "if" interface vary from vendor to vendor, but also from release to release from the same vendor in some cases. Further, it was important for our link-layer software to be passed a copy of the device-specific link-level header (e.g., the Ethernet version of our link-layer needed the Ethernet header to determine which logical link a packet was coming from), but most Ethernet drivers did not provide a way to pass such information through the "if" interface. To solve this problem, we were forced to modify some of the vendor-supplied physical layer drivers to bypass the "if" interface and communicate directly with our link layer software.

If at some level the Unix kernel could provide a uniform device-access interface that is independent of any higher level protocol, such a bypass would not be necessary.

Asynchronous Event Signalling. There must be a way to deschedule and reschedule a process from within the driver. We explored using the Unix signal mechanism to manipulate process context, but the behavior of signals varied from implementation to implementation so widely that we believed that they could not be used portably. Consequently, we were forced to translate essentially asynchronous events into synchronous context switches using *sleep* and *wakeup*. In other words, the driver must queue information for a process and then either wake it up (if it is asleep) or wait for it to try to go to sleep (when it could then be immediately reawakened). In either case, the asynchronous event must wait for the process to use a synchronous mechanism to schedule the correct context.

A uniform set of signal semantics and an accompanying kernel-level interface, would allow our protocol driver to naturally post asynchronous events to user-space processes.

Kernel Facilities That We Used

In building the LINCS-Unix driver, we had to make decisions about what services would be universally available both across Unix implementations and,

to a lesser extent, across various operating systems. The following is a list of some of the services that our driver requires of its host operating system and a brief rationale for each.

timeout/untimeout. In order to handle protocol timeouts for retries, give-ups, and the like, there must be a way to do timer-driven asynchronous function calls [12]. Also, efficiency is improved if a mechanism exists to cancel a previously scheduled function call.

Although the timeout/untimeout facility seemed to be universally available across Unix systems, we found that its interfaces were not entirely and that the untimeout facility in particular was typically somewhat inefficient. Since this facility is available in most Unix kernels, it should have a standard interface. Further, the need for an efficient mechanism to both set and cancel timers is important for efficient protocol implementation as pointed out by David Clark [13].

Driver interface. To pass information across the user-to-kernel boundary, the user-space half of the IPC system makes an *ioctl* call on a pseudodevice which we have defined as part of the driver. The *ioctl* mechanism was chosen because it allows an arbitrary data structure (or pointer to a data structure, depending on the implementation) to be easily passed into the kernel. We assumed that all systems built around the user-space/system-space paradigm would include such a facility, and that it would be uniform enough for us to mask the differences using the C preprocessor.

Some effort should be made to standardize the method of parameter passing between user space and system space to make the parameter parsing section of the driver more portable.

C language preprocessor functionality. Our source codes depend on certain C preprocessor features which we hope are available in most environments. Specifically, the portable LINCS-Unix driver depends on the conditional macro-expansion capability of the preprocessor, which is likely to be part of any Unix implementation, but which may not be found as part of other operating systems. There are indications, however, that this functionality will be included as part of the ANSI standard for the C language, so we feel justified in using it as part of a portable code [14].

System clock. Most reliable communication protocols require access to some kind of clock with varying resolution requirements. The Delta-t and Deltagram protocols used by LINCS, for example, require millisecond resolution for packet aging purposes [15].

The clock resolution was not standard across implementations, however, so we had to put some effort into converting the units of the local clock into a canonical unit. While it is necessary for systems to have a variety of internal clocks, a uniform clock interface with a standard resolution should be available from all kernels.

Conclusion

We identified the need for a portable communication system that we hoped to implement initially as a kernel driver for reasons of efficiency. We did not want, however, to preclude systems where kernel-level device drivers were not the norm or were extremely difficult to implement. Unfortunately, we found that while a great deal of work had been done to standardize the user interface to UNIX, interfaces to common kernel-level services remain woefully nonstandard, making true portability difficult to achieve. Further, efforts to enhance portability carried substantial performance costs. Therefore, we have since decided to switch to a set of standard protocols and interfaces (TCP/IP and sockets) for the transport through link layers because each vendor can afford to port, maintain, and optimize its own implementation.

It is not clear to us that the existing standard protocol implementations are well suited to the support of supercomputing; hence our desire to write our own protocol drivers. In the face of such a variety of kernel implementations, however, it does not seem possible to obtain both portability and the level of performance necessary to justify the effort. Since driver-level protocol implementations seem to be important for performance, it seems imperative that the interfaces to kernel subsystems be made as standard as possible.
We further recommend that a lightweight tasking facility be added to the standard UNIX kernel to support a more modular and extensible programming model for driver implementation.

Acknowledgements

The network driver was a product of equal effort by the author and Jed Kaplan (now working for IBM Watson Research). Dr. John Fletcher helped in the design and implementation of the portable network driver. In particular, he designed and implemented the part of the network driver which runs as a user-level library. Joe Requa first designed and implemented the version of the kernel tasking on which we based our system and which we have since modified. The Link layer codes for the different systems (currently somewhat less than completely portable) were written by Jim Holeman, Mark Gary, and Richard Ruef.

I would further like to thank Dr. Richard Watson for his insightful suggestions and liberal editing of this paper.

References

[1] R. H. Watson and J. G. Fletcher, "An Architecture for Support of Network Operating System Services," *Computer Networks* , vol. 4, pp. 33–49, 1980.

[2] J. E. Donnelley, "Components of a Network Operating System," *Computer Networks* , vol. 3, pp. 389–399, 1979.

[3] R. W. Watson, "Delta-t Transport Protocol: Features and Experience Useful for High Performance Networks," IFIP Workshop on Protocols for High-Speed Networks, Zurich, Switzerland, May 9–11, 1989.

[4] J. G. Fletcher, "Introduction to LINCS," *Tentacle*, April 1982–March 1983, Lawrence Livermore National Laboratory, Livermore Computer Center, Livermore, CA.

[5] R. H. Watson and S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems* , vol. 5 No. 2, pp. 97–120, May 1987.

[6] D.R. Cheriton, et al. , "Thoth, a Portable Real Time Operating System," *Communications of the ACM*, February 1979, pp. 105–115.

[7] R. C. Holt, *Concurrent Euclid, The Unix System, and Tunis.* Addison Wesley, 1986.

[8] D.R. Cheriton, and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," in *Proc. 9th Symposium on Operating System Principles*, October 1983, pp. 128–139.

[9] A. Tevanian et al., "Mach Threads and the Unix Kernel: The Battle for Control,", Carnegie-Mellon University, Department of Computer Science, Technical Report CMU-CS-87-149, August 1987.

[10] J. M. Bach, *The Design of the Unix Operating System.* Prentice Hall, 1986.

[11] *Unix System Managers Manual*, 4.3 Berkeley Software Distribution, Virtual Vax-11 Version (1986), p. SMM:15-4.

[12] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1989.

[13] D. D. Clark, "Protocol Performance—Why Networks Don't Go Fast," a tutorial on Internetworking presented at Interop, San Jose, Ca., October 1989.

[14] American National Standard for Information Systems (ANSI), Draft Proposed, Programming Language C, 1987.

[15] J. G. Fletcher and R. W. Watson, "Mechanisms for a Reliable Timer-Based Protocol," *Computer Networks* , vol. 2, pp. 271–290, 1978.

Richard Wolski
P.O. Box 808, L-60
Livermore, CA 94550

(415)423-8594

rwolski@lll-lcc.llnl.gov