Proceedings of the Second Workshop on Workstation Operating Systems, Pacific Grove, CA, Sept. 1989, pp 125-129

A Multi-User, Multi-Language Open Operating System

(extended abstract)

Michael L. Scott
Thomas J. LeBlanc
Brian D. Marsh

University of Rochester
Department of Computer Science
Rochester, NY 14627

April 1989

ABSTRACT

An open operating system provides a high degree of programming flexibility and efficiency. At the same time, it generally requires that all programs be written in a single language, and provides no protection other than that which is available from the compiler. These limitations may be tolerated by the user of a personal computer, but they become unacceptable on a workstation that must run untrusted software written in many different languages. Psyche is an operating system designed to make the most effective possible use of shared-memory multiprocessors and uniprocessor machines. It combines the flexibility of an open operating system with the ability to write in multiple languages and to establish solid protection boundaries. It also provides the efficiency of an open operating system for programs that do not require protection.

1. Introduction

An *open operating system* can be defined as one in which system services can be directly accessed, modified, and replaced by user-level code. More loosely (and less accurately), the term has come to be used for any operating system that operates without protection boundaries. The essence of openness is really a consequence of this lack of protection: independently-developed software modules may interact in a direct and easy fashion.

For a personal computer, openness offers two compelling advantages: flexibility and efficiency. Flexibility stems from the opportunity to modify, invoke, or build upon existing pieces of code. Efficiency stems from the lack of heavyweight context switches, data movement across narrow interfaces, or unnecessary layers of abstraction. Research results from the Pilot [4] and Cedar [6] projects at Xerox PARC testify to the usefulness of open systems. Clark's

This work was supported in part by NSF CER grant number DCR-8320136, Darpa ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

experience with Swift [2] testifies to their efficiency. Open systems have attracted the attention of language designers as well [1,7], and are an important part of the commercial market for personal computers.

Unfortunately, the flexibility and efficiency of open operating systems is obtained at a serious price. Protection is available only to the extent that it is provided by high-level language compilers. With multiple compilers (as on a commercial PC) there is no protection whatsoever. With a single available language there is little opportunity to use most pre-existing software. There is also no opportunity to employ programming styles or paradigms unsupported by the programming language. One is generally unable, for example, to program simultaneously in Cedar, Smalltalk, and Lisp within a single open system.

On a multiprocessor workstation, the use of a single language also constrains the choice of parallel programming models. A traditional closed operating system enforces the use of a single notion of process, communication, and sharing. An open operating system removes this restriction at the level of the operating system, but reinstates it at the level of the programming language. In a single-language system one cannot, for example, program simultaneously with actors, futures, parallel DO loops, and processes with monitors. One certainly cannot write a single program that uses different models in different sections of the code.

2. The Psyche Approach

Psyche [5] is an operating system under development at the University of Rochester. Its primary goal is to support efficient, flexible use of large-scale shared-memory multiprocessors. At the same time, it offers what we believe to be an ideal environment for small-scale multiprocessors and workstations. Specifically, Psyche combines the flexibility of an open operating system with the ability to write in multiple languages and to establish solid protection boundaries. It accomplishes this without forfeiting the performance available to programs in an open operating system.

Psyche can be described as an "open system by default." Its user-level programs occupy a uniform address space in which the sharing of code and data is natural and easy. Within this address space the user can, *if desired*, establish protection domains such that access across the boundary of a domain requires kernel intervention, with kernel checking of access rights. With compiler-enforced protection, no boundaries may be needed. On a workstation (particularly one that is programmed in multiple languages) boundaries can guarantee that mutually-suspicious programs are unable to damage each other.

The kernel itself is protected but minimal. It is the one part of Psyche that cannot be modified by users. Its principal purpose is to protect hardware resources and to establish

protection boundaries in user space. Most traditional operating system functions are provided outside the kernel. As in an open operating system, user-modifiable code is responsible for process creation, destruction, and scheduling, inter-process and network communication, local and remote file systems, program loading, and the management of virtual memory backing store.

Protection in Psyche is organized on the basis of data abstractions called *realms*. Each realm contains data and may contain code. Realms are created by invoking a kernel operation that returns an empty shell. In addition to filling in the contents of the shell, the creator can also specify that the realm should be used as the "root" of a protection domain, and that a given number of "virtual processors" should be devoted to executing processes in that domain. These virtual processors are known as *activations* of the domain. On a multiprocessor, they will be scattered across the nodes of the machine. The kernel timeslices between activations on each individual node.

Processes are the active entities that execute user programs. As in an open operating system a process can attempt to execute a subroutine anywhere in user space simply by addressing it. It is not guaranteed, however, that the access will succeed. Each process executes in a single protection domain at a time. If it attempts to touch a realm that is not in the *view* (i.e. the memory map) of the current protection domain, an access fault occurs and the virtual processor executing the process traps into the kernel. The kernel then checks access rights.

If the process does not have the right to access the target realm, the kernel informs the virtual processor that an error has occurred. If the process possesses a *key* that allows "optimized" access to the target realm, that realm is added to the memory map of the protection domain and the faulting instruction is restarted. We say that the realm has been *opened* for optimized access from this domain. If the process possesses a key that allows "protected" access to the target realm, the kernel *moves* the process. It finds the protection domain rooted by the target realm, informs some virtual processor of that domain that the process has arrived, and informs the current virtual processor that the process has departed.

Communication from the kernel to the virtual processors takes the form of signals, or *upcalls*, that resemble software interrupts. Upcalls occur when a process moves to a new protection domain, when it returns, and whenever an error occurs. In addition, user-level code can establish upcall handlers for wall time and interval timers, and can arrange to receive a warning in advance of virtual processor preemption.

The kernel keeps track of which processes are in which protection domains (in order to help them return after executing protected subroutines), but its help is not required to create or destroy processes or to perform context switches between them. When a virtual processor traps into the

kernel it is likely to be executing a different process than it was the last time it returned into user space. Processes in the same protection domain share the same memory map and are vulnerable to each other's mistakes. It is likely, therefore, that programmers will choose to implement them in a common style. In one domain, they may be appear as Cedar processes. In another, they may be Ada tasks. A Psyche process is likely to change representations as it moves between domains. Within a single domain, context switching between processes occurs entirely in user mode, and is as lightweight and efficient as a simple coroutine transfer.

If domains were entirely disjoint, Psyche would behave like a collection of open operating systems connected together by remote procedure calls. It is expected, however, that realms will frequently belong to more than one domain. A message buffer realm, for example, might be used for efficient communication between a pair of applications without exposing either to malicious or accidental damage by the other. To facilitate access to such realms, the Psyche kernel establishes a convention for synchronization. Domain-specific routines for blocking and unblocking processes can be associated with each virtual processor. When an operation on a shared realm must block, the current process can leave the address of its own unblock routine in a data structure of the realm.

By relying on a uniform address space, and by using the same jump-to-subroutine instruction to initiate both protected and optimized calls, Psyche has separated the issues of protection and performance from the semantics of realm operations. Users can determine which operations will be optimized and which will be protected simply by changing the distribution of keys or the contents of the access lists against which keys are matched. The flexibility and customizability of an open operating system can therefore be obtained without abandoning protection or the ability to program in multiple languages. The efficiency of an open operating system can be obtained in those cases where protection is not required.

3. Project Status

Our initial implementation of Psyche is written in C++ and runs on the BBN Butterfly Plus multiprocessor. We have completed the major portions of the kernel and are experimenting with user-level software. We have separated the machine-dependent and machine-independent portions of the implementation, so that Psyche can be ported to other machines. We are particularly interested in multiprocessor architectures such as the Berkeley SPUR, DEC Firefly, IBM ACE, or Xerox Dragon.

To facilitate kernel development we have implemented a remote, source-level debugger in the style of the Topaz TeleDebug facility [3]. The front end for the debugger runs on a Sun workstation and communicates via UDP and serial lines with a low-level debugging stub that

underlies the Psyche kernel. The low-level debugger was the first piece of the kernel to be written, and has proven extremely valuable.

In the context of large-scale multiprocessors, we are working with members of the computer vision and planning groups within our department to develop an integrated laboratory for real-time active vision and robotics. Psyche provides the foundation in this laboratory for high-level vision, planning, and robot control. Effective implementation of the full range of robot functions will require several different models of parallelism, for which Psyche is ideally suited. In addition, practical experience in the vision lab will provide feedback on the Psyche design.

References

[1] Brinch Hansen, P., *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[2] Clark, D., "The Structuring of Systems Using Upcalls," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 171-180. In *ACM Operating Systems Review 19*:5.

[3] Redell, D., "Experience with Topaz TeleDebugging," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pp. 35-44. In *ACM SIGPLAN Notices 24*:1 (January 1989).

[4] Redell, D. D. and others, "Pilot: An Operating System for a Personal Computer," *CACM 23*:2 (February 1980), pp. 81-92.

[5] Scott, M. L., T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, 15-19 August 1988, pp. 255-262, vol. II – Software.

[6] Swinehart, D., P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM TOPLAS 8*:4 (October 1986), pp. 419-490.

[7] Wirth, N., "From Programming Language Design to Computer Construction," *CACM 28*:2 (February 1985), pp. 159-164. The 1984 Turing Award Lecture.