Efficient, Portable, and Robust Extension of Operating System Functionality

Amin Vahdat, Douglas Ghormley, and Thomas Anderson
Computer Science Division
UC Berkeley

December 5, 1994

Abstract

Currently, operating systems are not chosen for underlying system features, but rather for the performance of the underlying hardware, available application programs, and system stability. Consequently, operating system vendors are reluctant to incorporate new operating system functionality since they risk both increased development time and decreased system stability. Previous efforts to make it easier for operating systems to incorporate new features have enjoyed only limited success because of performance bottlenecks or limited support for existing applications. This paper outlines a portable, efficient, and robust method for extending operating system functionality. Specifically, we propose building operating systems entirely as a library linked with every application using software-based fault isolation for protection. In order to demonstrate the validity of this technique, we are building an operating system which will provide global resource allocation in a network of workstations.

1 Introduction

Over the past several decades, a vast body of valuable operating system research has been conducted. Unfortunately, only a small percentage of this innovation has found its way into commercial operating systems. Even generally accepted ideas can take many years to appear in production systems. To combat this, operating system researchers spend a great deal of time focusing on research methodology: new ways of building operating systems to allow for faster assimilation of innovation into mainstream systems. To date, microkernel, application-specific, user-level, and portable operating systems have met with only limited success. However, we believe that the development of efficient software-based fault isolation will allow circumvention of many of the performance bottlenecks that has hampered the adoption of these previous efforts. We thus propose building an operating system as a layer on top of commercial systems to allow efficient, portable, and robust exploration and extension of operating system functionality.

Examples abound of operating system innovations which have gone largely ignored by commercial systems. A very limited sample of recent topics include: load sharing [Zhou et al. 1992], process migration [Theimer et al. 1985, Douglis & Ousterhout 1991], shared file system/virtual memory cache [Nelson et al. 1988], shared virtual memory [Li & Hudak 1989], multi-threading, fast user-level communication primitives [Bershad et al. 1990, von Eicken et al. 1992], upcalls [Clark 1985], network paging [Iftode et al. 1993], and parallel program support [Anderson et al. 1994]. Researchers have spent years speculating about why these ideas go largely ignored by industry. One possible explanation is that the ideas are not useful in the first place, so no one wants the features in their operating system.

Another, more plausible, explanation involves the motivation behind purchasing an operating system. Users typically do not purchase an operating system for the functionality or features of the system. Rather, an operating system is evaluated largely by the supported application programs, cost/performance of the hardware/OS combination, and system robustness. For example, the Apollo operating system implemented features which today remain state of the art, yet the system failed as a product for two reasons. First, the system was not UNIX-compatible so it did not support a large body of application programs, and second the hardware performance became uncompetitive. At the other extreme, DOS enjoys widespread popularity despite limited functionality; its popularity stems from a huge application base and inexpensive hardware.

Even when purchasing decisions are based on the operating system, the relevant issue is usually software reliability[1] rather than underlying features of competing systems. The lack of weight placed upon operating system functionality relative to system reliability only further discourages innovation since operating systems are notoriously brittle. As an example, most hard disk drives contain hardware to cache disk blocks and to schedule sector accesses. This functionality is provided in hardware despite the fact that Unix systems have more efficiently provided these services in software for years. The primary reason for this is that it is less difficult and more cost effective to implement this in hardware than to modify DOS to support such functionality.

Given that underlying operating system features are not very significant in purchasing decisions, vendors are reluctant to incorporate new technology. Unless an innovation provides significantly better application performance, enables the creation of application programs which were not previously feasible, or makes the system more robust, it is likely to be ignored. Even a fairly simple, effective idea such as the split virtual memory/file cache [Nelson et al. 1988] has only recently been added to commercial systems. New functionality can compromise both operating system functionality and robustness, and vendors are well aware that a delay in the introduction of an operating system can cost millions in lost hardware sales.

As a result of the difficulty associated with transferring research innovations to commercial systems, many research efforts focus upon new ways of building operating systems to facilitate technology transfer. In this position paper, we first discuss possible explanations for the limited success of previous efforts. Next, we propose a new approach to building operating systems that layers on top of commercial systems. Using this method, we believe that many new system features can be implemented quickly and portably while still maintaining good performance. As an initial prototype validating our methodology, we present GLUnix, a global layer operating system for a network of workstations.

2 Alternative Approaches

For years, researchers have striven for a design methodology that allows for easy extension of operating system functionality. Some of the earliest efforts were microkernel designs [Wulf et al. 1974, Accetta et al. 1986, Cheriton 1988, Mullender et al. 1990] which allow for user-level emulation of different operating systems. Despite efforts to make microkernel systems portable to different hardware platforms [Young et al. 1987], the required effort is still non-trivial. More importantly, the performance of such systems is limited by the costs of context switching, crossing protection boundaries, and inter-process communication. The emulation of standard operating systems such as UNIX or DOS through user-level servers further limits the performance of microkernel systems. Vendors are unlikely to accept design methodologies that significantly limit the available performance of the underlying hardware.

To achieve more rapid development time and greater portability, a number of operating systems have been built as user-level servers or libraries on top of commercial operating systems. Unfortunately, systems such as Eden [Lazowska et al. 1981], Condor [Mutka & Livny 1991], and PVM [Sunderam 1990] only provide partial solutions. Though some allow the execution of standard UNIX applications, they cannot do general purpose resource allocation without suffering the same inefficiencies found in microkernel systems. The resource allocator's code and data must be protected as a separate server requiring inter-process communication and context switches.

More recently, research efforts have focused on application-specific operating systems [Anderson 1992, Yokote 1992, Bershad et al. 1994, Engler et al. 1994]. While this approach appears promising, it is not intended as a framework for providing new system functionality, but rather as a way to allow applications to affect system policy. Unless significant speedup can be demonstrated for a wide variety of applications, vendors are unlikely to discard existing systems to reimplement their operating system from scratch. Even then, a methodology for extending operating system features and functionality is lacking.

Finally, a number of recent operating systems, such as Solaris and BSD 4.4, focus on portability. One of the goals of such systems is decoupling the choice of operating system from a particular hardware platform. Unfortunately, the time to port such systems to new hardware platforms remains significant. For example, the release of the DEC Alpha was released almost one year after the hardware was initially available because the port of OSF/1 to the Alpha proved more difficult than anticipated.

[1] As an example, reliability was a major selling point of SunOS versus initial versions of AIX.

3 A New Methodology

3.1 Full User-Level Functionality

We propose to bypass the limitations of the approaches outlined above by linking a protected operating system library with all application programs. The key enabling technology for such a system organization is software-based fault isolation (SFI) [Wahbe et al. 1993]. Traditional hardware protection can be efficiently implemented in software in a language-independent fashion by modifying the object code to insert checks before each store and indirect branch operation to catch addressing errors. Aggressive compiler optimization techniques reduce the overhead of these software checks to 3-7% on several contemporary RISC processors. Unrecompiled binary files can also be fault isolated, though the overhead is currently 20-25% [Lucco 1994]. In summary, SFI allows privileged operating system resource allocation code to execute in the application's address space with relatively low overhead.

Using SFI, we are able to build protected operating system functionality entirely at the user-level by dynamically linking the operating system library to each application. All system calls are redirected to procedure calls in the operating system library [Jones 1993]. Thus, a user-level virtual operating system layer is built using the underlying commercial system as a building block. Novel operating system functionality can be implemented more efficiently than in traditional systems since no hardware protection boundaries need be crossed—the new kernel code is invoked by a procedure call within the application's address space without the need for a kernel trap or a context switch. Any shared state needed to coordinate multiple user-level operating system libraries can be maintained using shared memory segments or interprocess communication primitives. The operating system also becomes more robust since errors in the user-level OS can be diagnosed using standard, widely-available debugging tools.

This approach faces two limitations. First, an arbitrary piece of kernel functionality cannot necessarily be implemented since the system is limited by the semantics and performance of the abstractions available at the user-level. However, as described below, a relatively rich set of features can be implemented at the user-level. The second potential pitfall is the overhead imposed by SFI. We believe that speedups gained from not having to cross protection boundaries and reduced system development time[2] will offset this slowdown.

3.2 GLUnix

To demonstrate the ideas outlined in this paper, we are building an operating system to perform global resource allocation in a network of workstations. Our system, GLUnix (Global Layer UNIX), glues together individual UNIX operating systems to provide a single system image of the machines in a network. GLUnix strives to make all resources in a network of workstations transparently available to each user. Thus, all of the idle processing power, memory, network capacity, and disk bandwidth in the network should be made available in a fair manner for both sequential and parallel applications. To provide such functionality, GLUnix must support coscheduling of parallel programs [Ousterhout 1982], idle resource detection [Mutka & Livny 1991, Arpaci et al. 1994], process migration [Theimer et al. 1985, Douglis & Ousterhout 1991], fast user-level communication [von Eicken et al. 1992, Martin 1994], remote paging [Iftode et al. 1993], and fault-tolerance [Borg et al. 1989].

Many of the features in GLUnix are not particularly novel. However, they have never been successfully implemented together in a coherent, usable system. GLUnix will enable efficient execution of parallel programs and improve the performance of memory or I/O intensive applications without requiring kernel modifications. With SFI as an enabling technology, we can thus build an efficient, portable operating system that does not rely on specific details of the native operating system. Currently, an initial version of GLUnix implementing much of the described functionality runs on HP and Sun workstations. The system is not yet fully distributed (decision making is centralized) and has not yet been integrated with SFI.

To ensure portability, GLUnix relies on a minimal set of standardized features of the underlying operating system. GLUnix should be portable to any system which supports inter-process communication, process signalling for scheduling and migration, and access to machine load statistics for idle resource detection. This design methodology should make porting GLUnix to new hardware platforms significantly easier than

[2] Since processors are improving in performance by 50% a year, faster system development time translates into faster total system performance.

any of the alternative methods discussed above.

4 Conclusions

This paper outlines a methodology for extending operating system functionality which maintains both performance and compatibility with existing applications. Specifically, the operating system can be built as a user-level library using the native system services as a building block. Software-based fault isolation provides protection of code and data within the address space and further redirects existing system calls (through binary translation) into the user-level kernel library as necessary. Superior system performance is made possible because every kernel operation is no longer required to cross a hardware protection boundary. Using this technique, a large class of operating system features can be explored and implemented more efficiently, portably, and robustly. We are building an operating system to do global resource allocation in a network of workstations to demonstrate the validity of the techniques proposed in this paper.

References

[Accetta et al. 1986] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pp. 93–112, June 1986.

[Anderson 1992] Anderson, T. The Case for Application-Specific Operating Systems. In *Proceeding of the Third Workshop on Workstation Operating Systems*, pp. 92–94, April 1992.

[Anderson et al. 1994] Anderson, T. E., Culler, D. E., and Patterson, D. A. A Case for NOW (Networks of Workstations). *IEEE Micro*, 1994. To appear in special issue.

[Arpaci et al. 1994] Arpaci, R., Dusseau, A., Vahdat, A., Liu, L., Anderson, T., and Patterson, D. The Interaction of Parallel and Sequential Workload on a Network of Workstations. Technical Report CSD-94-838, U.C. Berkeley, October 1994. Also submitted for publication.

[Bershad et al. 1990] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight Remote Procedure Calls. In *ACM Transactions on Computer Systems*, pp. 37–54, February 1990.

[Bershad et al. 1994] Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. G. SPIN—An Extensible Microkernel for Application-Specific Operating System Services. Technical report, University of Washingtion, 1994.

[Borg et al. 1989] Borg, A., Blau, W., Graetsch, W., Heermann, F., and Oberle, W. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–23, February 1989.

[Cheriton 1988] Cheriton, D. R. The V Distributed System. In *Communications of the ACM*, pp. 314–333, March 1988.

[Clark 1985] Clark, D. D. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 171–180, December 1–4 1985.

[Douglis & Ousterhout 1991] Douglis, F. and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, August 1991.

[Engler et al. 1994] Engler, D. R., Kaashoek, M. F., and O'Toole, J. W. The Operating System Kernel as a Secure Programmable Machine. MIT Technical Report, 1994.

[Iftode et al. 1993] Iftode, L., Li, K., and Petersen, K. Memory Servers for Multicomputers. In *COMPCON*, February 1993.

[Jones 1993] Jones, M. B. Interposition Agents: Transparently Interposng User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, December 1993.

[Lazowska et al. 1981] Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M., Fowler, R., and Vestal, S. The Architecture of the Eden System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 148—159, December 1981.

[Li & Hudak 1989] Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lucco 1994] Lucco, S. Personal communication, June 1994.

[Martin 1994] Martin, R. P. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of the 2nd Hot Interconnects Conference*, July 1994. Submitted for publication.

[Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Mutka & Livny 1991] Mutka, M. M. and Livny, M. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[Nelson et al. 1988] Nelson, M., Welch, B., and Ousterhout, J. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[Ousterhout 1982] Ousterhout, J. K. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pp. 22–30, May 1982.

[Sunderam 1990] Sunderam, V. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Theimer et al. 1985] Theimer, M., Landtz, K., and Cheriton, D. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 2–12, December 1985.

[von Eicken et al. 1992] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. (Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley).

[Wahbe et al. 1993] Wahbe, R., Lucco, S., and Anderson, T. Efficent software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 203–216, December 1993.

[Wulf et al. 1974] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–344, June 1974.

[Yokote 1992] Yokote, Y. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 414–434. ACM, October 1992.

[Young et al. 1987] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63–76, November 1987.

[Zhou et al. 1992] Zhou, S., Wang, J., Zheng, X., and Delisle, P. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.