Issues in the Design of an Extensible Operating System

Stefan Savage and Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

June 28, 1994

Abstract

Extensible operating systems are designed around the principle that a system can be dynamically customized to best serve application needs. However, realizing this goal in a safe and efficient manner poses a number of unique problems. In this paper, we examine the requirements for constructing robust extensible systems and discuss implementation techniques to satisfy those requirements with low overhead.

1 Introduction

An extensible operating system is composed of a set of interfaces and implementations that can be changed as needed by an application or a set of applications. The design of an extensible system reflects the position that the core services and performance requirements of all applications cannot be met entirely in advance by any operating system. Consequently, an extensible operating system must provide a software infrastructure into which new components can be installed as though they were part of the native system.

In general, operating systems have attempted to provide extensibility through some sort of protection mechanism, such as capabilities [Levy 84], IPC [Black et al. 92, Rozier et al. 88], or special compilers, linkers, or interpreters [Mogul et al. 87, Wahbe et al. 93]. In practice though, the protection mechanism *by itself* has proven insufficient to support general extensibility. For example, a powerful IPC mechanism, although necessary for communicating with a file server in user space, is not sufficient for changing the semantics or performance characteristics of that file server without large amounts of effort [Druschel 94]. The server, and the environment in which the server operates, must be amenable to change, which an IPC mechanism permits, but does not guarantee.

A protection mechanism is also not sufficient for dealing with the many problems that can occur when potentially untrusting and untrustworthy subsystems cooperate to provide a service. Although the mechanism may prevent corruption at the level of the memory system, it cannot prevent higher levels of corruption which occur when modules do not satisfy their specified requirements. For example, a segmented memory architecture may ensure that an extension never makes an illegal memory reference, but it cannot automatically mediate access to a shared data structure, or guarantee completion of an extensible interrupt handler.

This research was sponsored by the Advanced Research Projects Agency under Arpa Order No. 7597, by the National Science Foundation under the terms of a Presidential Young Investigator Award, by the Office of Naval Research under the terms of the Young Investigator Program, and by a grant from the Xerox corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the University of Washington, the Advanced Research Projects Agency, the National Science Foundation, the Office of Naval Research, Xerox, or the U.S. Government.

We are currently building an extensible operating system in which we are attempting to address the problems raised by the need for generalized extensibility [Bershad et al. 94]. In our system, application programs dynamically extend the set of services provided by the operating system by installing new code at runtime. In designing our system, we have found that an extensible system must at once satisfy three goals:

- Incrementality. Small changes to the system's behavior can be affected with small amounts of code [Kiczales et al. 91].
- Correctness. An extension provided by an application should not compromise the overall integrity of the system.
- Efficiency. The potential for an extension should incur no overhead. The use of an extension should have an overhead which is determined by the extension's code, and not the infrastructure that has enabled the extension.

In this paper, we explore the issues that arise in the design and implementation of an extensible system. We show that many of the problems in an extensible system, which arise because of the need for incrementality, are solved by design, and not mechanism. We also show that, in cases where a mechanism is applicable, there there is no single piece of technology ("silver bullet") that can be applied everywhere. Lastly, we describe how a language/compiler framework can make it possible to apply efficiently a range of protection mechanisms for code in an extensible system. The essence of our discussion is that the language provides a veneer which can conceal much of the system's state from extensions thereby protecting it, and that the compiler can efficiently protect that state which cannot be entirely concealed from an extension. In this way, the responsibility for ensuring system integrity moves from the extensions, of which there are many, to their compiler, of which there is one.

2 What is an extensible system?

An extensible operating system provides a set of mechanisms and interfaces that allow applications to tailor system behavior to specific demands. For example, the filing component of an extensible operating system might allow applications to customize the buffer cache replacement algorithm. Similarly, a disk driver can be replaced with an "intelligent" disk driver that applies some application-specific processing (such as decompression or search) on a data block only for files on which the processing is required. A communication protocol component might provide interfaces that allow applications to customize the windowing and acknowledgment timeouts for a specific kind of network or load. More trivially, in an extensible system, a communication protocol such as SLIP can be installed when the protocol is first used, rather than when the system is first configured.

All operating systems are in one sense or another extensible. Given the source code, or a binary interface, any user can write new operating system code for an application, build it into a new kernel, install, and reboot. Of course, few users are actually capable of negotiating through system build procedures [ODE 91], let alone writing new code for the system [Leffler et al. 89, Leffler & McKusick 91, Pietrek 93].

Moreover, security and integrity concerns generally make arbitrary user extensions unacceptable, either on a personal computer [Livingston 94], on a shared machine, or even a machine on a shared network. For example, in our shared computing environment, Macintosh users are not allowed to share the same physical network as Unix users because of a history of network problems introduced by Macintosh extensions. Similarly, machines on which non-staff members have privileged accounts are not generally permitted to access shared NFS servers without special consideration. Moreover, ours, which is a fairly progressive and lax computing environment, is substantially more flexible than most others. As a result of these kinds of technical and administrative prohibitions, demanding applications, such as database, distributed transactions, parallel processing, and multimedia, are forced to rely on the "out of the box" interfaces and implementations when others may be more appropriate for reasons of performance, portability, interoperability, or data integrity [Kiczales et al. 93, Engler et al. 94].

3 Incrementality and interfaces

A key issue in the design and use of an extensible system is the ease with which changes to system behavior can be applied, or its *incrementality*. Ideally, small changes to system behavior can be expressed with small amounts of additional code. Incrementality is primarily a function of a system's design characteristics, rather than any underlying mechanism or framework on which the design is based. For example, IPC, protected procedure calls or capabilities do not guarantee that small changes to a file system, external pager, or device driver can be made with small amounts of effort. Some frameworks however can simplify the construction of an incremental interface. Most notably, object-based and delegation-based interfaces have been shown to facilitate small changes [Andert 94, Khalidi & Nelson 93] when extensions are trusted. Their utility with untrusted extensions remains in question, however as they do not provide a complete solution [Steinberg 94].

Incrementality is not a natural by-product of a system's overall gross structure. Recently, for example, there have been efforts to migrate code from the operating system into a user's application directly through the use of operating system libraries [Maeda & Bershad 93, Thekkath et al. 93]. Library-level networking protocols place the bulk of the BSD UDP/IP and TCP/IP protocol stacks into a library that can be linked with the application. One of the anticipated advantages of this structure was that applications could easily make small changes to the protocol implementation in order to achieve high performance. The problem, of course, is that the protocol stacks, when originally written, were not themselves designed to be extensible. Simply moving them into another protection domain space did not change this design characteristic.

The lack of incrementality in the protocol implementation also aggravated the migration of the protocols out of the operating system and into applications. The protocol needed to be partitioned in such a way that the operating system retained responsibility for protocol initialization and termination, with the application handling packet transfer. There was no natural "cut point" in the implementation to facilitate this partitioning. Instead, the designers were forced to introduce an entirely new protocol into the operating system in order to extend it. The purpose of this new protocol was to mimic the existing protocols during connection management, but to leave undefined the routines which were implemented in the library. While this structure ultimately worked, the unconventional protocol splice was the source of many bugs in the initial implementation. In contrast, a protocol architecture such as the x-kernel [Hutchinson & Peterson 88], which allows designers to construct a protocol graph from small components, exhibits substantially more incrementality.[1]

Interfaces to hardware

An important class of software interfaces in an extensible system are those that provide access to the raw hardware. For example, an extensible system might provide a standard interface to hardware services such as TLB management, device management, interrupts, traps, and CPU allocation. Interfaces that encapsulate hardware functionality define the limits of a system's extensibility because all higher level interfaces necessarily depend on them. However, interfaces to hardware resources are only appropriate for the incremental extension of low-level hardware services. For example, extending a file system to support data compression cannot be easily accomplished in terms of basic device access interfaces.

Interfaces to software

Interfaces to software resources such as file systems, network protocols, or process management do not naturally exhibit incrementality. The UNIX *Vnode* interface [Kleiman 86] offers an example of an interface that exhibits some, but not complete, incrementality. With it, it is possible to define an entirely new file system without having to change the disk device driver or the system's exported file system interface. On the other hand, extending an existing file system implementation built on top of the Vnode interface to use a different buffer cache manager, or even to modify the buffer cache manager's block replacement algorithm, is not trivial, as Vnode was intended to encapsulate a whole file system, not just pieces of one.

Mach's external pager interface offers another example of an interface that exhibits some, but not complete, incrementality [Young et al. 87]. Using the interface, applications can provide their own pager without

[1] The x-kernel was not used in this project because the goal was to compare performance of the best available kernel-based and library-based protocol implementations in a variety of configurations.

having to rewrite the entire virtual memory system. However, if an application is only concerned with determining if a particular page is presently in core, then it must either provide its own external pager in entirety, or it must rely on a different underlying virtual memory interface [McNamee 94].

3.1 Extensibility namespaces

Fundamentally, incremental extension requires fine grained decomposition of an implementation such that the structure provides convenient "hooks" for extension code. Together, these hooks define the namespace into which extensions can be installed. The size of this namespace determines a system's extensibility. If the namespace is empty, that is there are no places into which code can be plugged, then the system has limited extensibility. We know of no system in which the namespace is entirely empty. As stated earlier, it is always possible to change something, provided sufficient resources. In contrast, if the extensibility namespace is large, that is, there are many places where new code can be merged with old code, then the system can be extended easily.

MS-DOS, which is perhaps the most successful operating system (in terms of units shipped) offers no boundaries to extensibility. New code can be installed after the system is running. Applications can modify operating system data structures, trap and interrupt handlers, and even kernel code. Anything needed by an application can be added if not already there. This freedom though comes at a substantial cost: any application can crash another application, or the system; applications that work in isolation may cease to work when run in combination; and the lack of any defined interface to either operating system services, or operating system extensibility, has created a system configuration nightmare [Draves 93]. We address this issue in the next section.

4 Correctness

An extensible system provides a namespace into which existing operating system code and new extensions can be introduced. In the case of MS-DOS, a single, shared address space containing all user and system code offers that namespace. For microkernel-based systems, such as Mach, Chorus and Amoeba, the namespace is composed of interfaces implemented in separate address spaces using an interprocess communication facility. For some UNIX systems, which provide a limited form of extensibility through downloadable device drivers, this namespace is created through a combination of protected files and the kernel's address space.

Although a common namespace enables extensions, it introduces the possibility of system corruption as one extension is able to name, and possibly effect, the resources of another. Because system extensions can have independent and unrelated origins with varying degrees of trust and correctness, the system must view them as potentially hostile entities. For example, if an extension which runs as an interrupt handler never terminates then this is a failure to meet liveness constraints. If two extensions share data then this may lead to corruption if ordering and type correctness are not guaranteed. If one extension writes over the code or data used by another, this too is a failure of system integrity. There are many other opportunities for extensions to violate a system's trust, such as:

- an extension enters an infinite loop
- an extension deadlocks on itself or another extension because it fails to follow a locking protocol,
- an extension fails to lock/unlock a shared resource when modifying it,
- an extension calls out into an exported system interface with inappropriate parameters,
- an extension invokes an illegal operation on object,
- an extension invokes a legal operation on an object that no longer exists,
- an extension fails to relinquish a shared resource that must be preempted and reallocated to another context.

The system must resolve the trust issues which surround such extensions before they can be safely integrated into a running system. For example, an extension in an infinite loop can be preempted, ensuring that it will not monopolize the processor.

In general, there are two ways by which an extension can be assured correct: *verification*, which ensures that an extension is correct before it is ever installed, and *protection*, which attempts to ensure that an extension behaves correctly after it is installed, and bounds the damage done if it does not.

4.1 Verification

Verification may involve software verification techniques that statically prove the correctness of a piece of code [Wing 90] or simply an administrative decision that the code is trusted. Software verification techniques do not provide a complete or general purpose solution because the complexity of programs that can be entirely verified this way is restricted. However, automatic verification can be useful when applied to small code fragments in conjunction with runtime protection mechanisms.

Most systems use a form of verification based on manual certification together with a high degree of pre-existing trust. System functionality is collected into a single protection domain, called a "kernel," within which code is assumed to be correct. Someone with sufficient administrative authority examines a proposed extension, determines that it is correct, and installs it into the system.

The quality of verification based on manual certification, or "faith in shrink-wrapped software" is limited, though. This type of verification works best when there are relatively few extensions which originate from a single administrative domain, such as the operating system's primary vendor, as with the dynamically loadable device drivers found in many Unix systems. Systems which have many extensions from independent sources, such as MS-DOS or the Macintosh OS, tend to have robustness problems when they use this method of verification.

4.2 Protection

In the absence of verifiably correct system extensions, it is necessary to impose constraints on an extension at runtime. For example, a memory reference operation can be constrained to manipulate memory within the allocated domain of the extension. The constraint itself can be enforced by a mechanism, for example, memory protection, which in turn has any one of a number of implementations, such as hardware memory management, software fault isolation [Wahbe et al. 93, Hastings & Joyce 92] or language support [Mogul et al. 87].

Ensuring correctness strictly with protection mechanisms is not straightforward. For each system interface exposed to an extension, be it a hardware or software interface, it is necessary to define the constraints governing the use, but preventing the misuse of that interface. Then, given a constraint, it is necessary to determine an appropriate mechanism which enforces that constraint. Finally, given a mechanism, it is necessary to choose an implementation of the mechanism which is correct, and which has good performance.

Table 1 enumerates several areas in which protection in an extensible system must be addressed. Each area is problematic because it exposes a vulnerable interface to untrusted code. For each exposed interface, the table shows a constraint governing that interface's use, a mechanism enforcing that constraint, and a set of possible implementations for the mechanism. The implementations are representative of those that have been used in systems in the past.

The table shows that mechanisms which provide memory protection are necessary but not sufficient for building an extensible system. The key problem is that inappropriate loads and stores are not the only way to keep a system from working. Instead, memory protection only ensures a program's microbehavior. Macrobehavior, that is "what does a code segment do" as opposed to "what does it reference" is much more complicated to guarantee. The table also shows that no one mechanism enforces all constraints, and that a variety of implementations exist for a single mechanism.

Protection by design

Table 1 only describes vulnerable areas for which it appears reasonable to rely on mechanized protection facilities. Some operations, though, cannot be protected by general mechanisms. Specifically, operations which are constrained by abstract state requirements are difficult to isolate through mechanical means alone. For example, preventing an extension from sending misaddressed IP packets to a network device is

Area	Interface	Constraint	Mechanism	Example Implementations
Memory Protection	Load/Store	R/W allowed	Memory ref checking	Hardware MMU Software fault isolation Typesafety
	Invocation	Call allowed	Protected jump	Protected procedure call (hardware) Protected procedure call (software) Typesafety
Synchronization	Shared data	Synchronized	Atomicity	Locks Wait-free data structures Transactional memory
Liveness	Invocation	Boundedness	Preemption	Hardware interrupts Compiler-inserted yields Compiler-verified boundedness

Table 1: *This table shows several areas for which correctness is an issue in an extensible system.*

difficult to accomplish in the same mechanical fashion as is used to enforce memory protection. Run-time assertions explicit in the code are required for these types of interfaces.

Ensuring that such assertions are complete is difficult, suggesting that extensible interfaces be designed to minimize their need. One strategy that appears promising is to define stateless interfaces that impose no constraints on the caller. As a motivating example from the distributed systems domain, the NFS interface, which is stateless, does not rely on the correct functioning of clients [Sandberg 85]. In contrast, the interface to the Sprite file system, which is not stateless, does rely on the correct functioning of clients [Nelson et al. 88].

With a stateless interface, the client of the interface (an extension) is responsible for maintaining all state and for passing pieces of it across the interface as needed. Typesafety can be used to ensure that the data passed across an interface is correct with respect to its definition type. The statelessness of the interface ensures that the order and frequency of calls across the interface cannot violate the interface's integrity. We expect that some, but not all interfaces can be constructed this way. For example, we have defined stateless CPU management and scheduling interfaces that allow applications to integrate a custom threads package into the system scheduler.

5 Automating protection

A fundamental goal in the design of an extensible system is to keep the set of interfaces that are "protected by design" to a minimum. This requires that as much of the protection facilities as possible be automated. The nature of the implementations described in the previous section suggests that the language and the compiler offer an appropriate path to automated protection.

Automation through language and compiler design offers two important benefits. First, that information which cannot be safely revealed to an untrusted extension need not be. The compiler can hide and manage this state together with a trusted runtime system. For example, shared data in a concurrent system frequently requires the use of a synchronization protocol to mediate access to the shared resource and ensure its consistency. In conventional systems this is implemented by the programmer using mutual exclusion operations such as *lock* and *unlock*. With an untrusted extension, the lock may be taken out of order, taken not at all, or never relinquished. The compiler can apply mechanisms such as locks, wait-free synchronization or transactional memory which relieve the programmer from the responsibility of correctly managing synchronization state [Herlihy 90, Barnes 93]. For example, if the compiler can determine that a code segment accessing a shared data structure executes in bounded time, it can insert a locking protocol into the code sequence. The compiler can also cooperate with the run-time system to ensure that extensions execute for bounded periods of time [Hutchinson 87].

A second advantage of automation using the language and compiler is that it facilitates static enforcement of the protection mechanisms, that is, it can verify code. For example, a compiler can statically prevent an extension from accessing a variable outside the extension's scope, enabling memory reference protection with no runtime cost. This implies that extensions should be written in a typesafe language, shifting much of the memory protection enforcement from run-time to compile-time. With typesafety, only array bounds checking is required to enforce memory reference protection. Typesafety can be also be used to constrain (at compile-time) the semantic correctness of assignment and parameter passing, relieving the provider of an interface from the responsibility (and penalty) of checking in many cases.

5.1 Difficulties with a language-oriented approach

There are two potential downsides to relying on compiler and language support to enforce protection. First, legacy code cannot be used. We are, however, unaware of any technique, be it hardware or software, which allows an an untrusted, arbitrary piece of legacy code to be safely used as an extension in an operating system considering the problem areas shown in Table 1.

Second, static analysis of a typesafe language is not a complete solution to the protection problem. Runtime processing and overhead is impossible to avoid when checking references on array bounds. An alternative is to rely on hardware memory protection mechanisms, which can provide inexpensive coarse-grain protection. However, hardware protection mechanisms can incur a significant cost when changing protection domains, even when heavily optimized [Bershad et al. 90]. In contrast, domain switching with software methods can be inexpensive [Wahbe et al. 93]. This suggests that software mechanisms for isolating memory references are most effective when they protect code:

- which is frequently invoked (because then the invocation overhead is low),
- which executes for a short period of time (because then the absolute cost of protection is low).

Fortunately, this desired balance is nicely achieved by the needs of an extensible operating system. Applications can run in their own protection domain and be subjected to hardware protection mechanisms. They can install code fragments as extensions, subjected to software protection mechanisms, into the operating system. The size of the imposed extension, relative to the core system services, is likely to be small either because it represents an incremental extension to another service, or because it provides a control and data conduit to a larger extension that is itself protected with hardware mechanisms [Cao 94, Yuhara et al. 94]. Moreover, this structure allows programmers to write the bulk of their code in the programming language of their choice; only extensions need to be written in a typesafe language.

6 Conclusion

In this paper, we have discussed some of the important concerns that arise when building an extensible system. *Incrementality* determines the ease of introducing an extension. *Correctness* bounds the degree of integration achievable. *Efficiency* dictates how incrementality should be implemented. The last two issues are well matched by the capabilities of compiler technology. Compilers allow run-time costs to be reduced through static verification, while also reducing design costs or opportunities for error by hiding vulnerable shared state.

References

[Andert 94] Andert, G. Object frameworks in the Taligent OS. *Proceedings of the Spring 1994 COMPCON Conference*, February 1994.

[Barnes 93] Barnes, G. A method for implementing lock-free data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms & Architecture*, June 1993.

[Bershad et al. 90] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Bershad et al. 94] 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. In *Proceedings of the 1994 European SIGOPS Workshop*, September 1994. To appear.

[Black et al. 92] Black, D. L. et al. Microkernel Operating System Architecture and Mach. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[Cao 94] Cao, P. Personal communication. May 1994.

[Draves 93] Draves, R. The Case for Run-Time Replaceable Kernel Modules. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 160–164, 1993.

[Druschel 94] Druschel, P. Efficient Support for Incremental Customization of OS Services. Technical report, Department of Computer Science, University of Arizona, 1994.

[Engler et al. 94] Engler, D., Kaashoek, M. F., and O'Toole, J. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the 1994 European SIGOPS Workshop*, September 1994. To appear.

[Hastings & Joyce 92] Hastings, R. and Joyce, B. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, January 1992.

[Herlihy 90] Herlihy, M. A Methodology for Implementing Highly Concurrent Structures. *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, March 1990.

[Hutchinson & Peterson 88] Hutchinson, N. C. and Peterson, L. L. Design of the *x*-Kernel. In *Proceedings of the SIGCOMM '88 Symposiumy*, pages 65–75, August 1988.

[Hutchinson 87] Hutchinson, N. C. *Emerald: An Object-Based Language for Distributed Programming.* PhD dissertation, University of Washington Dept. of Computer Science and Engineering, January 1987. Department of Computer Science technical report 87-01-01.

[Khalidi & Nelson 93] Khalidi, Y. A. and Nelson, M. N. An Implementation of UNIX on an Object-Oriented Operating System. In *Proceedings of the 1993 Winter USENIX Conference*, pages 469–480, January 1993.

[Kiczales et al. 91] Kiczales, G., des Riviéres, J., and Bobrow, D. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kiczales et al. 93] Kiczales, G., Lamping, J., Maeda, C., Keppel, D., and McNamee, D. The Need for Customizable Operating Systems. In *Proceedings of the Fourt Workshop on Workstation Operating Systems*, pages 165–169, October 1993.

[Kleiman 86] Kleiman, S. R. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the 1986 USENIX Conference*, pages 238–247, 1986.

[Leffler & McKusick 91] Leffler, S. J. and McKusick, M. K. *The Design and Implementation of the 4.3BSD UNIX Operating System Answer Book.* Addison-Wesley Publishing, 1991.

[Leffler et al. 89] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing, 1989.

[Levy 84] Levy, H. M. *Capability-Based Computer Systems.* Digital Press, 1984.

[Livingston 94] Livingston, B. Window manager: Black screen of death. *InfoWord*, May 1994.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[McNamee 94] McNamee, D. Supremo: Increaing the Flexibility and Performance of Virtual Memory. Technical Report UW-TR-94-07-01, University of Washington Dept. of Computer Science and Engineering, 1994.

[Mogul et al. 87] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[Nelson et al. 88] Nelson, M. N., Welch, B. B., and Ousterhout, J. K. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[ODE 91] Open Software Foundation. *ODE Manula Pages*, November 1991.

[Pietrek 93] Pietrek, M. *Windows Internals.* Addison-Wesley Publishing, 1993.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[Sandberg 85] Sandberg, R. Design and Implementation of the Sun Network Filesystem. *Proc. USENIX 1985 Summer Conf. USENIX Association*, pages 35–50, 1985.

[Steinberg 94] Steinberg, S. G. Corba. *Wired*, 2(7), 1994.

[Thekkath et al. 93] Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[Wing 90] Wing, J. M. A Specifier's Introduction to Formal Methods. *IEEE Compter*, 23(9):10–22, September 1990.

[Young et al. 87] 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 *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[Yuhara et al. 94] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.