The Operating System Kernel as a Secure Programmable Machine

Dawson R. Engler M. Frans Kaashoek James W. O'Toole Jr.

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

To provide modularity and performance, operating system kernels should have only minimal embedded functionality. Today's operating systems are large, inefficient and, most importantly, inflexible. In our view, most operating system performance and flexibility problems can be eliminated simply by pushing the operating system interface lower. Our goal is to put abstractions traditionally implemented by the kernel out into user-space, where user-level libraries and servers abstract the exposed hardware resources. To achieve this goal, we have defined a new operating system structure, *exokernel*, that safely exports the resources defined by the underlying hardware. To enable applications to benefit from full hardware functionality and performance, they are allowed to download additions to the supervisor-mode execution environment. To guarantee that these extensions are safe, techniques such as code inspection, inlined cross-domain procedure calls, and secure languages are used. To test and evaluate exokernels and their customization techniques a prototype exokernel, *Aegis*, is being developed.

1 Introduction

Traditionally, operating systems have implemented system objects (such as processes, address spaces, exception handling), forcing applications to make do with what is provided. Frequently these abstractions are inadequate, their implementation slow, or both [4, 5, 14, 15, 22, 29, 31]. We believe that these problems can be solved by simply lowering the interface to the hardware that is enforced by the kernel: namely, by exporting physical resources to applications directly, management and abstraction of these resources can be specialized for simplicity, efficiency, and appropriateness. Exposing the raw hardware aids kernel extensibility in a direct manner: the easiest kernel primitive to extend is the one that is not there. The low-level interface exported by the exokernel opens the possibility of user-level implementations of basic system objects. Furthermore, these implementations can coexist on the same system allowing them to be tuned to a specific application domain and, potentially, realized at a level of efficiency unmatched by any current system. More importantly, the exokernel allows applications to craft their own abstractions (without emulation on high-level OS primitives) in ways that are fundamentally not possible in any other operating system.

Unlike our approach, a traditional monolithic operating system normally provides a single fixed policy for resource management. However, no single policy is best for all applications. For example, file system and relational database applications often have different storage access patterns and therefore benefit from different buffer and disk management policies. Similar policy conflicts arise in almost every area of resource management (for example, in virtual memory management [15]). Microkernels have pushed some policy decisions out of the kernel into user-level servers. However, these policies cannot, generally, be altered or replaced by non-privileged applications. Additionally, in these microkernels the supervisor mode still defines a high-level, fixed interface among the kernel, user-level services, and applications. Therefore, applications suffer from a large number of protection domain crossings. These crossings are typically expensive, as they in-

{engler|kaashoek|james}@lcs.mit.edu, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 617-253-7436. This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985; the second author is partially supported by an NSF National Young Investigator award. 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 U.S. government.

volve entering supervisor mode and changing address spaces [3, 7, 25].

The disadvantages of fixed operating system policies are made worse by the growing diversity in hardware platforms and rapid advances of technology. For example, in a mobile computer it is of crucial importance to save power by carefully managing the spin-down policy of the disk. As an example of rapid advances in technology, new architectures support protected user-level communication that requires little involvement of the operating system [6, 19]. By fixing policies operating system software prevents applications from evolving with the hardware.

By moving management of resources out of the operating system kernel, an exokernel allows aggressive customization. We see three reasons for customization:

- Tailor policies to given hardware configurations, allowing applications to advance with the hardware.
- Tailor policies to given applications. For example, the interaction of application and the virtual memory system is crucial to the performance of applications. Changes to VM policies can enable distributed shared memory systems and garbage collectors to run an order of magnitude faster.
- Improve performance by reducing the overhead of interrupts, TLB faults, traps and context switches, which are some of the primary sources of overhead in today's operating systems. Customization allows compile-time and run-time specialization.

To minimize the performance impact of crossing the kernel boundary, the exokernel allows applications to run user code safely in the kernel. In this paper we discuss three techniques to make the exokernel design safe and efficient: (1) code inspection, (2) inlined cross-domain calls, and (3) type-safe languages. We are prototyping the new design and implementation techniques in *Aegis*, an experimental exokernel.

The outline of the rest of the paper is as follows: Section 2 presents a design sketch of a prototype exokernel, *Aegis*; Section 3 discusses our proposal for a secure programmable machine and the implementation techniques that permit safe customization across the supervisor mode barrier. In Section 4, we briefly discuss related work and in Section 5 we conclude.

2 An Exokernel Structure

We are building the *Aegis* kernel to test the feasibility of an exokernel that provides minimal resource allocation, arbitration services, and a programmable interface. In designing the *Aegis* exokernel, we intend to strictly limit the functionality that is embedded and non-customizable. Many of the primitives supplied by other "minimalistic" kernels are viewed as applications by the exokernel. For example, we do not expect the kernel to include page-table management, a file-system, inter-process communication, or processes management. We expect the *Aegis* system to enable unprecedented experimentation with techniques for safe customization of application operating environments.

The exokernel's sole function is to allocate, deallocate, and multiplex physical resources in a secure way. The resources exported by the exokernel include physical memory (divided into pages), the CPU (divided into time-slices), the TLB, context-identifiers, and disk memory (divided into blocks).

Access to physical resources is controlled through 64-bit capabilities that rely on "security though obscurity" [23]. [1] The operational view of access control is that each resource (e.g., time-slices, pages) is associated with a capability. Every point the resource could be used is protected by a guard. When an application wishes to use a resource it presents the capability to the guard. The guard checks the access rights given by the capability.

The exokernel performs two non-physical functions: exception forwarding and upcalls. The exokernel hands exceptions directly to applications, enabling efficient user-level exception handling. Upcalls implement synchronous cross-domain transfer of control. Operationally, an upcall transfers the program counter in one domain to an agreed upon value in another, and installs the called domain's exception context. This mechanism is light-weight (our unoptimized implementation takes 20 instructions on a MIPS R3000) and lets applications build their own IPC semantics (i.e., a client can trust a server to save and restore registers).

To allow applications to fully exploit the physical hardware, the exokernel provides facilities to download code behind the supervisor barrier. For example, this code can access unmapped physical memory, which can be important for handling TLB-miss exceptions. Additionally, by allowing user code and state to reside behind the supervisor barrier, the frequency of user/supervisor kernel crossings can be reduced. The techniques we use for this are described in Section 3.

The exokernel's low-level interface is defined by

[1] Since we intend for the exokernel to scale across different trust models, the capability size can be modified. For instance, in an embedded system that has cooperating processes, capabilities may simply be the name of the object.

the raw hardware: physical memory pages, DMA channels, I/O devices, translation look-aside buffer, processors, addressing context identifiers, and interrupt/trap events. This "abstraction free" interface allows flexible user-level implementations of traditionally rigidly defined OS services. For instance, since the exokernel does not enforce a particular page-table structure, applications can aggressively specialize them. Since the address space of a "protected object" may contain just a few pages, it can be mapped using a small linear page-table, while a single 64-bit address-space subsystem may use inverted page-tables. As another example, the exokernel allows applications to directly manipulate context-identifiers, enabling efficient protection domain switching, an action important to many applications [32].

3 A Secure Programmable Machine

We expect the operating system kernel to provide a programmable machine that permits applications to safely and efficiently manage system resources. The secure machine executes in supervisor mode because it uses privileged instructions. Our design protects the secure machine against unsafe application code using traditional methods: unsafe application code executes only in separate protection domains.

In addition, to permit efficient management of system resources, our design permits controlled execution of application code in supervisor mode. This makes the secure machine programmable and enables safe, efficient customization of the operating system. However, only safe application code may be executed in supervisor mode. We are examining several implementation techniques that permit safe customization across the supervisor mode barrier. These techniques ensure that application code will not access privileged instructions or protected data in uncontrolled ways:

1. **Code Inspection.** Code introduced into the secure programmable machine must be free of privileged instructions. As shown by Deutsch [11], machine code can be inspected to guard against wild loads, stores, and jumps. Execution time can also be controlled by bounding loop iteration counts. With sufficiently stringent restrictions on the application code, it can be safely executed in supervisor mode, and primitives of the secure machine that use privileged instructions can be inlined into the inspected code.

2. **Inlined cross-domain calls.** Code inspection that prohibits indirect loads and stores will be too restrictive for many application purposes. Wahbe [32] shows that sandboxing can be used to restrict the domain of indirect memory operations, but with an overhead of several instructions per guarded operation. The motivation of sandboxing is to simulate address-space protection domains when cross-domain calls are expensive. However, if the application code is executed in supervisor mode, then switching address spaces may require only a few privileged instructions (e.g., to change the processor context identifier). The application can execute in supervisor mode, but in a restricted addressing context. Protected data structures can be manipulated by calling machine primitives. If wild jumps are prevented, then cross-domain calls can be safely inlined, including the instructions that change the addressing context. An example of where this is used is in a TLB exception handler, which must be invoked with low-latency after a TLB miss.

3. **Type-safe language.** If applications need to customize the operating system kernel with operations that require frequent and intensive manipulation of shared protected data structures, then page-granularity protections may not suffice. Using a type-safe language and a trusted compiler might enable closer integration of application and kernel code, while simultaneously increasing portability. An example of this is our packet-filter engine, which uses dynamic code generation [12] to directly compile packet-filter predicates specified in a declarative, type-safe language.

We expect each of these techniques to be useful for some kinds of exokernel extensions, depending on the tradeoffs involved.

4 Related Work

Many early operating systems papers discussed the need for extensible kernels [34, 17, 21, 33]. Lampson's description of the CAL-TSS [20] and Brinch Hansen's microkernel paper [13] are two classic rationales. Hydra was the most ambitious system to have the separation of kernel policy and mechanism as one of its central tenets [34]. Modern revisitations of microkernels have also argued for kernel extensibility [16, 26, 1, 28, 30, 10]. Anderson [2] and Kiczales et al. [18] also recently argued for minimalism and customization. Two important differences between our work and previous approaches are that: (1) the fixed kernel interface in these systems is a high-level one (e.g., page-tables are implemented by the kernel) and (2) customization is expensive since it is usually accomplished through the addition of servers that interact with the kernel via IPC.

The interface provided by the VM/370 operating system [9] is very similar to what would be provided by an ideal exokernel, namely the raw hardware. However, the important difference is that VM/370 provides this interface by *virtualizing* the entire base-machine. Since this machine can be quite complicated and expensive to emulate faithfully, virtualization can have a high cost in both kernel complexity and efficiency. In contrast, the exokernel *exports* hardware resources rather than emulating them, allowing an efficient and fast implementation.

Synthesis is an innovative operating system that has inspired some of our work [22]; for example, the use of runtime code generation. The *Aegis* system, however, uses a portable runtime code generation system [12], while the synthesis kernel is written in assembly, and uses ad-hoc runtime code generation macros to specialize system calls. Furthermore, the *Aegis* system will use runtime code generation to efficiently execute application kernel extensions.

Concurrently with our work the SPIN [5] and Cache Kernel [8] projects are also investigating adaptable kernels that allow applications to make policy decisions efficiently.

The SPIN system encapsulates policies in *spindles* that can be dynamically loaded into the kernel. To ensure safety, spindles will be written in a pointer-secure language and will be translated by a trusted compiler. In contrast, we anticipate using a variety of customization techniques that permit safe execution of small pieces of application code within the exokernel, some of which will allow the kernel interface to be extended using existing application code. In addition, the exokernel aggressively moves code out of the operating system kernel into user-space; its kernel interface provides a much low-level interface than microkernels do. The SPIN project, on the other hand, concentrates more on moving user code into a microkernel.

The Cache Kernel shares our belief that a low-level interface is the panacea that can cure current operating system troubles. The main difference between the two projects is in how low the operating system interface is pushed. The Cache Kernel designers still see themselves in the business of *abstracting* physical resources (i.e., hiding information, disallowing fine-grain user-management) rather than *exposing* them (i.e., removing as much abstraction/virtualization as possible). For example, the Cache Kernel provides address spaces, threads, and application kernels as kernel-based abstractions. In contrast, the exokernel exposes the machine primitives necessary to construct these abstractions to applications; direct manipulation of physical resources allows applications to implement services that were formerly enforced by the operating system (e.g., page-table management), with a corresponding improvement in flexibility and performance. Additionally, the Cache Kernel has an alarmist view about downloading application code into the kernel [8]; this restricts the efficiency and flexibility of their approach.

Techniques for certifying the safety of code include inspection and sandboxing. Deutsch showed how to safely introduce user-written instrumentation code into a running kernel by bounding loads, stores, jumps and runtime [11]. Making existing binaries safe has been done by Wahbe [32]. Although this technique will be useful for controlling entry points, we expect that loads and stores can be constrained more efficiently by changing the address context quickly within supervisor mode.

The use of a typesafe language to allow user applications to augment the operating system has been explored in a personal workstation environment [27]. Myers argued for using a type safe language to load application code into the object manager in a persistent object system [24].

5 Conclusions

Operating systems must shed the burden of their traditional policy-laden services. We suggest exokernels as an new minimal operating system design. The key exokernel idea is to obtain flexibility through a low-level kernel interface; i.e., the easiest kernel primitive to extend is the one that is not there. To minimize the overhead of protection domain crossings we proposed three customization techniques.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. *Proc. Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[2] T.E. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.

[3] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Proc. Fourth International Conference on ASPLOS*, 1991.

[4] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. Fourth International Conference on ASPLOS*, pages 96–107, Santa Clara, CA, April 1991.

[5] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. Spin - an extensible microkernel for application-specific operating system services. TR 94-03-03, Univ. of Washington, February 1994.

[6] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *The 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[7] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[8] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. *Proceedings of the Sixth SIGOPS European Workshop*, September 1994.

[9] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. Research and Development*, 25(5):483–490, September 1981.

[10] H. Custer. *Inside Windows/NT.* Microsoft Press, Redmond, WA, 1993.

[11] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. *Information Processing 71*, 1971.

[12] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[13] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[14] J.H. Hartman, A.B. Montz, David Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communication-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.

[15] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. *Proc. of the Fifth Conf. on Architectural Support for Programming languages and Operating Systems*, pages 187–199, October 1992.

[16] D. Hildebrand. An architectural overview of QNX. *Proc. Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[17] D.H.R. Huxtable and M.T. Warwick. Dynamic supervisors — their design and construction. *Proceedings of the First ACM Symposium on Operating Systems Principles*, 1967.

[18] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. In *Fourth Workshop on Workstation Operating Systems*, pages 165–170, October 1993.

[19] J. Kuskin et al. The Stanford FLASH multiprocessor. *The 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[20] B.W. Lampson. On reliable and extendable operating systems. *State of the Art Report*, 1, 1971.

[21] B.W. Lampson and H.E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[22] H. Massalin. *Synthesis: an efficient implementation of fundamental operating system services.* PhD thesis, Columbia University, 1992.

[23] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[24] A.C. Myers. Resolving the integrity/performance conflict. In *Fourth Workshop on Workstation Operating Systems*, pages 156–160, October 1993.

[25] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. *20th Annual International Symposium on Computer Architecture*, pages 27–38, 1993.

[26] R.F. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[27] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[28] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[29] M. Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, July 1981.

[30] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[31] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[32] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[33] B.A. Wichmann. A modular operating system. *Proc. IFIP Cong. 1968*, 1968.

[34] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessing operating system. *Communications of the ACM*, 17(6):337–345, July 1974.