Exterminate All Operating System Abstractions

Dawson R. Engler   M. Frans Kaashoek
{engler, kaashoek}@lcs.mit.edu
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

The defining tragedy of the operating systems community has been the definition of an operating system as software that both multiplexes and *abstracts* physical resources. The view that the OS should abstract the hardware is based on the assumption that it is possible both to define abstractions that are appropriate for all areas and to implement them to perform efficiently in all situations. We believe that the fallacy of this quixotic goal is self-evident, and that the operating system problems of the last two decades (poor performance, poor reliability, poor adaptability, and inflexibility) can be traced back to it. The solution we propose is simple: complete elimination of operating system abstractions by lowering the operating system interface to the hardware level.

1 Introduction

Throughout the history of computer science there has been a fairly constant opinion that "current" operating systems are inadequate [4, 7, 9, 11, 15, 18]. The literature is rife with specific examples that describe the cost of the inappropriate, inefficient abstractions peddled by operating systems [2, 4, 12, 13, 18, 23, 24]. This situation has persisted for the last three decades, and has survived numerous assaults (object-oriented operating systems and micro-kernels are two of the more popular movements). As a general rule, a concept that cannot be realized after such a long period of time should be reexamined.

The standard definition of an operating system is software that securely multiplexes and *abstracts* physical resources. We believe that this definition, specifically its view of the OS as an abstractor of hardware, is crippling and wrong. The basic intuition behind our arguments is that no OS abstraction can fit all applications. This is not a characteristic that is just a release away: it is *fundamentally* impossible to abstract resources in a way that is useful to all applications and to implement these abstractions in a way that is efficient across disparate needs. For some reason, OS implementors have decided that this simple law does not apply to them and that, indeed, they are compelled to indulge in the wholesale abstraction of hardware resources. In this position paper we concentrate on the following characteristics of operating systems that have been built in this manner: they are complex and large, which decreases system reliability and aggressively discourages change; they are overly general, which makes their use expensive and makes their implementation consume significant fractions of machine resources; and they enforce a high-level interface, which precludes the efficient implementation of new abstractions outside of the OS.

We believe, therefore, that the attempt to provide OS abstractions is the root of all operating system problems. We contend that these problems can be solved directly by the systematic elimination of OS abstractions, lowering the interface enforced by the OS to a level close to the raw hardware. It is important to note that we *favor* abstractions, but they should be implemented outside the operating system so that applications can select among a myriad of implementations or, if necessary, "roll their own".

The structure of this paper is as follows: Section 2 expounds our ideology; Section 3 outlines specific features of our target operating system structure; and Section 4 explores some of the advantages of this new OS structure and some common misconceptions in such a structure. We discuss related work in Section 5 and conclude in Section 6.

This work was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and by a NSF National Young Investigator Award.

2 The Jeremiad

For this paper, we define the operating system as any piece of software that the application cannot either change or avoid. User-level device drivers, privileged servers, and kernels are all included by this definition. The goal of the OS designer should be to push the interface this defines to the level of the raw hardware. We define *application-level* software as software that can be changed and/or avoided by any application; this is in contrast to software at *user-level* (or in *user-space*), which may require very high privileges to adapt or replace (e.g., replacing a device driver often requires "root" privileges). Much of the fixation micro-kernels have with putting pieces of the kernel into user-space comes from a confusion between user and application level.

The thesis of this position paper is that the operating system should not abstract physical resources. What the OS should do is what no other piece of software can do: safely multiplex physical resources. The motivation for this decision can be placed in the context of the "end-to-end" argument [21]: OS abstractions are reduntant or of little value when compared to the cost of providing them. We explore these issues more thoroughly below:

Poor reliability Abstracting resources (e.g., providing a full-featured virtual memory system with copy-on-write, memory-mapped I/O and other treats) requires a large amount of complex, multi-threaded code. These characteristics, along with dynamic storage allocation and management and the paging of kernel data structures and code, greatly decrease the reliability the system.

Poor adaptability The OS is large and complicated. Changing large, complicated pieces of software is *hard*. This creates a disincentive to incorporate new features or tune existing ones. Furthermore, since all applications "depend on" the OS, change is not localized. This provides an additional discouragement to adapting the OS implementation. Finally, only the kernel architect can incorporate new changes, which further restricts adaptability. The effects of this can be seen directly: how many of the good ideas in the last 10 SOSP conferences have been incorporated (or allowed at application-level) by any operating system other than the one they were developed on? For example, what operating systems support multiple protection domains within a single-address space, efficient IPC, or efficient and flexible virtual memory primitives?

Poor performance OS abstractions are often overly general, as they provide any feature needed by any reasonable application and all applications must use a given OS abstraction. Applications that do not need this feature pay unnecessary overhead [1, 18]. In the case of garbage collectors or database systems this cost can amount to an order of magnitude. Additionally, simply using a given feature is costly, since time must be spent selecting from a myriad of options [18]. Furthermore, the mere existence of OS abstractions consumes significant amounts of main memory, cache space, TLB space, and cycles, which could be used by applications to perform useful work.

Finally, any OS implementation makes trade-offs: whether to use a hierarchical or inverted page-table, whether to optimize for frequent reads or random writes, whether to have copy-on-write or a large page size, etc. Unfortunately, any trade-off penalizes applications that were not anticipated or neglected by the OS designer. However, this situation is easily avoidable: if the OS does not abstract resources, it does not have to such make trade-offs.

Poor flexibility The poor reliability, poor adaptability, and poor performance of operating systems would be acceptable if applications could just ignore the operating system and implement their own abstractions. Unfortunately, the high-level nature of current operating system interfaces makes this approach infeasible. The best that applications can do is emulate the desired feature on top of existing OS abstractions; unfortunately, such emulation is typically clumsy, complicated, and prohibitively expensive. For example, once the application has no access to the raw disk interface, database records must be emulated on top of files. The list of such examples is painfully long and continues to grow [2, 4, 13, 18, 23, 24].

In short, operating systems are complex, fragile, inflexible, and slow, because they have dabbled in the practice of providing a general purpose virtual machine. The operating system is basically hardware masquerading as software: it cannot be changed, all applications must use it, and the information it hides cannot be recovered. Operating system designers should learn what hardware designers learned a decade ago during the transition from CISC to RISC: hardware should provide primitives, *not* high-level abstractions.

3 The Solution: Eliminate OS Abstractions

We contend that the solution to all of these difficulties is straightforward: eliminate operating system abstractions. The OS should only export physical resources in a secure manner; it should not be in the business of presenting a pretty, machine-independent interface to applications.

In this section we give a general sketch of an OS structure that embodies a "abstraction-free", low-level interface. We call such a structure an *exokernel*. The sole function of an exokernel is to allocate, deallocate, and multiplex physical resources in a secure way. The resources exported by this kernel are those provided by the underlying hardware: physical memory (divided into pages), the CPU (divided into time-slices), disk memory (divided into blocks), DMA channels, I/O devices, translation look-aside buffer, addressing context identifiers, and interrupt/trap events.

Security is enforced by associating every resource usage or binding point with a guard that checks access privileges. For example, as one of the steps in preserving memory integrity, the kernel guards the TLB by checking any virtual-to-physical mappings given by applications before they are inserted into the TLB.

The kernel can optimize global performance by its control over the allocation and revocation of physical resources. With this control it can enforce proportional sharing, or what resources are allocated to which domains.

To make these examples concrete, we detail what address spaces, time-slices, and IPC might look like under the regime we have described. The details we present are highly machine-specific, but the general outline should be similar across machines; the main goal in each is to answer the question: what is the minimum functionality that the kernel needs to provide in order for this primitive to be implemented in application space?

Address space To allow application-implemented virtual memory, the OS must support bootstrapping of page-tables, allocation of physical memory, modification of mapping hardware (e.g., TLB), and exception propagation. The simplest bootstrapping mechanism is to provide a small number of "guaranteed mappings" that can be used to map the page-table and exception handling code. Physical memory allocation should support requests for a given page number (enabling such techniques as "page-coloring" for improved caching [5]). Privileged instructions (e.g., flush, probe, and modify instructions) can be wrapped in systems calls, and those that write to privileged state (e.g., TLB write instructions) are associated with access checks. Exception propagation is done in a direct manner by (perhaps) saving a few scratch registers in some agreed-upon location in application-space and then jumping to an application-specified PC-address [24].

Of course, all of these operations can be sped up by downloading application code into the kernel [4, 9] or using a "software TLB" [14, 3] to cache translations. These implementation techniques aside, the full functionality provided by the underlying hardware should be exposed (e.g., reference bits, the ability to disable caching on a page-basis, the ability to use different pagesizes, etc.).

Process The only state that the operating system needs to define a process is a set of exception program counters that the operating system will jump to on an exception, an associated address space, and both prologue and epilogue code to be called when a time-slice is initiated and expires. Placing context-switching under application control (through the application-defined prologue and epilogue code) enables techniques such as moving the program counter out of critical sections at context-switch time [6].

IPC The basic functionality required by IPC is simply the transfer of a PC from one protection domain to an agreed-upon value in another, with the donation of the current time-slice, installation of the called domain's exception context, and an indication of which process initiated the call. This extremely lightweight, synchronous, cross-domain calling mechanism implements the bare-minimum required by any IPC mechanism, allowing the application to pay for just the functionality that it requires. For example, a client that trusts a server may allow the server to save and restore the registers it needs, instead of saving the entire register file on every IPC. Since the machine state of current RISC machines is growing larger [19], this can be crucial for good performance.

This is far from a complete enumeration of all system objects (for example, we neglect disks and devices), but should give a feel for what level of functionality the OS is *required* to provide. The bare minimum is much removed from the policy-laden, overly general, and restrictive implementations surrounding us today. (A more complete discussion and evaluation of the exokernel methodology and a prototype exokernel can be found in Engler [10]; our prototype exokernel performs 10-100 times faster than a mature monolithic system in operations such as as IPC, exception forwarding, and virtual memory manipulations.)

4 Discussion

We discuss how our proposed structure solves the traditional problems of reliability, efficiency, and extensibility; these points have at their core the simple principle that the most efficient, reliable, and extensible OS abstraction is the one that is not there.

Reliability Exposing hardware resources safely and efficiently requires neither sophisticated algorithms or many lines of code. As a result, the operating system can be small and readily understood: both of these properties aid correctness.

Additionally, the application-level implementation of operating system services is likely to be much simpler in structure and smaller in realization than a traditional, general-purpose OS. For example, it does not have to multithread among multiple malicious entities, or worry about peculiar characteristics of supervisor mode (e.g., the particular locking constraints that arise within the kernel to guard against loss of interrupts and deadlock). Finally, since this application OS "trusts" the application, it can use application state directly and simply; a general-purpose OS is constantly in the business of copying user data, guarding against illegal addresses, and checking for validity. All of these concerns can be ignored in an application-level OS since, if the application does something wrong, the damage is to itself only.

Adaptability The kernel's simplicity enables easy modification. Furthermore, since most of the operating system code is used simply to track ownership and access rights, there is not much that needs to be tuned.

By allowing application-level implementations, we have removed the dependence of the entire system on the implementation. In other words, by localizing experiments within a single operating system subsystem (library or server), applications that wish to use the new feature can link it in. Those that do not, do not need to. Traditional operating systems occupy the unfortunate position of having every application depend on their correct and appropriate implementation. This "depends on" relationship drastically limits the degree to which experimentation can be carried out and the results used.

A more prosaic example of improved adaptability is that the implementation can now occur in application space, with access to user-development environments (e.g., debuggers and profilers).

Efficiency Resource management has been put into application space, allowing implementations to exploit application-specific knowledge in making tradeoffs (e.g., optimizing for reads or random writes, sparse address spaces, etc.); furthermore, since implementations can be highly specialized, they can eliminate the cost of generality present in most OS abstractions. From an engineering standpoint, this structure allows a broad pool of non-kernel architects to implement alternative implementations. Since the entire system does not depend on these implementations, application-level operating systems can be readily experimented with and altered by non-privileged implementors; furthermore, these operating systems will be readily used, since they do not have to be used by the entire system (and hence trusted in a very real sense). Finally, since all operations can occur in the same address space, the current contortions to minimize the cross-domain costs of TLB pollution/misses, system call traps, and context-switches are completely obviated.

Flexibility Applications can now implement system objects in ways that are fundamentally impossible on traditional operating systems. Radical page-table structures, process abstractions, address spaces, and filesystems can be constructed safely and efficiently on top of this structure. We expect that this freedom will enable a broad class of applications that are not feasible under current operating systems.

The ability of application-level operating systems to support powerful, efficient, and unusual abstractions cannot be overemphasized. By allowing any application writer to implement fundamental system objects, the degree, ease, and pervasiveness of experimentation and *utilization* of the results of this experimentation can dramatically increase.

We discuss some frequently asked questions about the methodology we have proposed.

Won't executables become large? As a practical matter, libraries that implement traditional abstractions can be sizable. However, shared libraries can be used to combat this problem. They have been used successfully in equivalent situations. For example, the X-window libraries are typically dynamically linked. In the worst case, servers can be used to multiplex code, data and threads of control.

Doesn't this cause portability problems? There are two levels of portability that must be provided, machine portability and OS interface portability. The first may be achieved in the standard manner: namely, a low-level layer that hides machine dependence. The second can be achieved through application-level implementations of industry standards (e.g., POSIX). The important difference is that the implementation of these layers are now in application space and, therefore, can be replaced without special privileges, simplifying the addition and development of new standards and features not anticipated by kernel architects.

What happens to the system structure when any application can define its own interfaces? As language, GUI and standard library implementors can attest, preventing a Babel of incompatible interfaces is simple: define standards and conventions. As we argue in this position paper, the effects of hardwiring high-level interfaces into the system structure are a convincing demonstration that such an approach is not the right way to define a standard. In closing, while an exokernel allows the possibility of a chaotic system it also allows the creation of a harmonious, elegant one as well: a system whose structure does not have to be anticipated by kernel architects.

How can system state be shared? Trusted (or at least highly accountable) servers can be used to manage shared, fault-isolated caches of system objects such as file-buffers. This methodology is has been explored in the context of microkernels; many of the lessons learned are directly applicable to exokernels.

5 Related Work

Micro-kernels were originally intended to solve many of the problems we have listed. Unfortunately, they have floundered for a number of reasons. First, while they allow replacing of device drivers and high-level servers, such operations typically can only be done by trusted applications. Second, they are still in the business of providing a virtual machine to applications. The high-level interface that they enforce precludes much of the experimentation that we desire (the reader is invited to compare the primitives described in Section 3 to current micro-kernels). Third, their rigid interface tends to be rudimentary when compared to their monolithic counterparts; they often achieve simplicity by implementing only a small set of high-level OS abstractions. For example, a micro-kernel may have achieved simplicity by dropping support for `mmap`, memory-mapped I/O, and full-featured virtual memory, but not given alternative mechanisms to implement the functionality these features provided: micro-kernels can give applications even less control over hardware resources than a monolithic system does.

Microkernel architects have realized that an operating system should be small; the crucial mistake they have made is in determining how an OS *should* get this way. It should become small not by enforcing a limited set of high-level operations, but instead, through the systematic elimination of all operating system abstractions in order to expose the hardware to application-level software; from this primal mud, applications can craft their own abstractions, chosen for appropriateness and efficiency, rather than make do with abstractions force-fed under duress. With micro-kernels, applications can have even fewer options than with monolithic ones.

Two current OS research efforts, the Cache Kernel [7] and Aegis [9, 10], adhere closely to our precepts for a model operating system. Further experience is needed to see if a low-level kernel interface is indeed the panacea that can cure current operating system troubles.

The open operating system for a single-user machine is motived by similar observations as the ones that motivate the exokernel [16]. However, the approach taken to extensibility taken is different. The exokernel's main task is secure multiplexing, while in the open operating system protection is not an issue at all, since it relies on the fact it is designed for a single-user machine. In addition, the exokernel attempts to define no OS abstractions, while in the open operating systems the file system and communications are standardized. Despite these differences, one can view the exokernel architecture as an instance of an open operating system.

The interface provided by the VM/370 OS [8] is very similar to what would be provided by our ideal OS: 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 result in a complex and inefficient OS. In contrast, our approach *exports* hardware resources rather than emulates them, allowing an efficient and fast implementation.

OS extensibility has a long history [15, 20]. Current attempts include SPIN [4], Bridge [17], and Vino [22]. Some of the techniques used in these systems, such as type-safe languages and software fault-isolation [25], are also applicable to exokernels. The commercial world has long looked at this issue in the form of unsafe dynamically loaded device drivers.

6 Conclusions

Two decades ago, Lampson summarized the state of the art; unfortunately, his characterization is still apt:

> A considerable amount of bitter experience in the design of operating systems has been accumulated in the last few years, both by the designers of the systems which are currently in use and by those who have been forced to use them. As a result, many people have been led to the conclusion that some radical changes must be made, both in the way we think about the functions of operating systems and in the way they are implemented [15].

We believe that these problems can be solved by 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 then be specialized for simplicity, efficiency, and appropriateness.

The low-level interface of our proposed OS structure would have allowed the bulk of operating system research in the last two decades to have been done easily and safely in application space. Furthermore, the impact of this research could have been much greater, since the implementation of its ideas could have been localized to specific applications.

References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. Thirteenth Symposium on Operating System Principles*, pages 95–109, October 1991.

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

[3] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on OSDI*, pages 243–253, June 1994.

[4] 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.

[5] B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS-VI*, 1994.

[6] B.N. Bershad, D.D. Redell, and J.R. Ellis. Fast mutual exclusion for uniprocessors. In *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 223–237, October 1992.

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

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

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop*, September 1994.

[10] Dawson R. Engler. The design and implementation of a prototype exokernel operating system. Master's thesis, MIT, 545 Technology Square, Boston MA 02139, February 1995.

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

[12] 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.

[13] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on ASPLOS*, pages 187–199, October 1992.

[14] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

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

[16] B.W. Lampson and R.F. Sproull. An open operating system for a single-user machine. *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 98–105, 1979.

[17] Steven Lucco. High-performance microkernel systems (abstract). In *Proc. of the first Symp. on OSDI*, November 1994.

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

[19] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer Usenix*, pages 247–256, June 1990.

[20] 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.

[21] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *Proc. of the Fifth SOSP*, pages 509–512, 1981.

[22] Margo Seltzer et al. An introduction to the architecture of the VINO kernel, November 1994.

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

[24] C. A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Sixth Conf. on ASPLOS*, 1994.

[25] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of Fourteenth SOSP*, pages 203–216, 1993.