Extensible Kernels are Leading OS Research Astray

Peter Druschel, Vivek S. Pai, Willy Zwaenepoel
Rice University
Computer Science Department
{druschel | vivek | willy} @cs.rice.edu

Abstract

We argue that ongoing research in extensible kernels largely fails to address the real challenges facing the OS community. Instead, these efforts have become entangled in trying to solve the safety problems that extensibility itself introduces into OS design. We propose a pragmatic approach to extensibility, where kernel extensions are used in experimental settings to evaluate and develop OS enhancements for demanding applications. Once developed and well understood, these enhancements are then migrated into the base operating system for production use. This approach obviates the need for guaranteeing safety of kernel extensions, allowing the OS research community to re-focus on the real challenges in OS design and implementation. To provide a concrete example of this approach, we analyze the techniques used in experimental HTTP servers to show how proper application design combined with generic enhancements to operating systems can provide the same benefits without requiring application-specific kernel extensions.

1. Introduction

Extensible operating systems are viewed by many as a technology that promises to solve some of the long-standing challenges in OS design. Among these challenges are (1) achieving performance close to the hardware's capabilities across a wide range of applications without giving up safety, (2) facilitating the rapid deployment of new OS innovations, and (3) containing and managing the ever-increasing complexity of OS implementations. We feel that skepticism is in order with regard to these claims, for two main reasons. First, the numerous research projects in extensible kernels have to date only produced performance-related results (challenge 1), and even in this category they have not yet convincingly demonstrated the benefits of extensibility. Projects like SPIN, Aegis, and VINO show performance improvements on a small set of benchmark applications that use kernel extensions [2, 6, 16]. Unfortunately, it is unclear to what extent the performance gains are due to extensibility, rather than merely resulting from optimizations that could equally be applied to an operating system that is not extensible.

Second, extensible kernels (i.e., those that allow the execution of untrusted, application-specific code in Turing-complete languages) introduce a number of new, hard problems into OS design. While prototype extensible kernels have not yet produced convincing evidence of the benefits of extensibility, they have clearly demonstrated the need for sophisticated techniques to deal with the unique safety problems introduced by extensibility. Such technologies include type-safe languages, trusted compilers, garbage-collected kernels, trusted dynamic linkers, capabilities, sandboxing, sophisticated resource management, proof-carrying executables, and even kernel-level transactional facilities [2, 6, 12, 14, 16, 18].

In addition to safety concerns, extensible kernels also raise difficult questions with respect to compatibility, interoperability, and evolution of OS implementations. In conventional systems, applications and operating system interact only through the API. This API is a contract that protects the application developer's investments by ensuring interoperability and backward compatibility, while still giving the OS developer maximum freedom in designing and evolving an implementation of the API. In an extensible system, applications can interact with the OS through additional extension interfaces. This introduces a difficult tradeoff. If the extension interfaces are not part of the contract, then applications that use these interfaces risk losing interoperability and compatibility with future OS releases. If they are part of the contract, then these interfaces constrain the implementation and evolution of the OS implementation. Moreover, the richer the extension interfaces—thus providing powerful extension capabilities—the more constrained the OS implementation becomes.

While these unique needs of extensible kernels create a host of new research opportunities for the operating system and programming language communities, along with prospects for fame and glory (funding, publications, and dissertations), it is unclear how solving these problems moves

the community any closer to solving the original challenges outlined above. In fact, the additional complexity required for safe extensibility aggravates the problem of managing and controlling the complexity of OS implementations.

Due to the difficulty of supporting extensibility that is powerful, safe, and practical from a software engineering standpoint, we do not expect that extensibility will find its way into commercial operating systems in the foreseeable future. On the other hand, extensible kernels can play a significant role in advancing the state of the art in operating systems. They can do so by stimulating research into optimizations that benefit particular classes of demanding applications, and those optimizations can be migrated to production operating systems. To understand how this technology transfer can take place, we must understand what sorts of issues can be explored in such an environment. We contend that additions and changes to the kernel can be broadly classified in two ways—*enhancements* and *customizations*.

An *enhancement* is simply an optimization of an existing service or the addition of a new interface and service. Enhancements may benefit only a certain class of applications, and they make at worst no difference to other applications. Since enhancements are generic, they can be provided or certified by a trusted entity, and no special kernel mechanisms are required to ensure their safety. In contrast to enhancements, *customizations* must be specified and/or selected on a per-application basis, because they may not be suitable for all applications. If they were enabled by default, they might actually harm the performance or jeopardize the correctness of certain applications. Customizations fall into two categories: those that we consider *inherently safe* (specified in a parametric or otherwise restricted language), and general extensions (specified in a Turing-complete language). It is only the latter category that concerns us, since this type of extension requires the additional, safety-related complexity in extensible kernels. For example, application-controlled file caching [4] and programmable packet filters [11] are inherently safe customizations. They are safe either by only allowing the application to select from a set of pre-defined options, or by using a constrained extension language that rules out potentially unsafe issues like looping, dynamic memory allocation, and the ability to retain state between invocations.

We argue that the real value of extensible kernels lies in their ability to stimulate research by allowing rapid experimentation using general extensions. We think that many (if not all) of these experimental extensions, once properly understood, can be recast as either enhancements or inherently safe customizations, and then migrated into commercial operating systems. This approach avoids most of the difficult issues raised by extensible kernels, while preserving most of its benefits. Operating systems experts in the research community can develop, prototype, and demonstrate OS features using extensible kernels. Once a feature is well understood and its benefits have been demonstrated, there is sufficient market push to drive OS vendors to incorporate it into their commercial systems as enhancements or inherently safe customizations. Since extensible systems are used only in the experimental or prototype stage, many of the difficult safety issues they raise in a production environment can be avoided. The difficult work of developing kernel extensions is done by experts in the research and advanced OS development communities. Finally, production operating systems need not be burdened with complex machinery to ensure safe extensibility and one can avoid the software engineering issues raised by extension interfaces.

To summarize our argument, we contend that extensible kernels will remain a tool for experimentation and prototyping. This realization implies that extensible system researchers should redirect their efforts towards meeting the challenges they originally set out to meet. Thus, we should give up on trying to make extensibility safe in a production environment, and focus instead on using extensible kernels as vehicles for new development that can help meet the real challenges facing OS design and implementation. Efforts in building extensible kernels should focus on mechanisms that allow flexible and powerful addition and interposition of kernel extensions, such as SPIN's dynamic binding framework [14]. Once in place, such kernels should then be used to support research into generic OS enhancements and inherently safe customizations that advance the state-of-the-art in OS design and implementation. The next section provides a concrete example of how a set of apparently application-specific extensions can be recast in the form of enhancements that do not require general kernel extensibility.

2. Analysis of WWW Servers

HTTP [1] servers (also called webservers) have been one of the benchmark applications used by supporters of extensible kernels. However, a close examination of the extensions used to support webservers suggests that most of the benefits could equally be achieved by generic enhancements to the base operating system. By surveying techniques used in publically-available web servers [15, 5] and in experimental designs [8], we have identified eight areas of interest where OS performance can impact webservers. They are: TCP implementation performance, TCP protocol optimizations, forking/switching overhead, filesystem performance, VM/cache control, data copying costs, double-buffering, and TCP checksum calculation. Commercial operating systems and webservers have already addressed some of these, while others have been addressed solely in the research community, with the use of extensible kernels. We contend that an analysis of these items shows

that extensible kernels are not needed to efficiently support webservers. For each item, the two relevant questions are "Can it be solved through better application design?" and "If not, can it be solved through enhancements and inherently safe customizations rather than general kernel extensions?" The first five problem areas have been (or can be) addressed by known techniques for better application design or application-independent kernel optimizations.

TCP implementations - Until recently, many commercial implementations of TCP were relatively inefficient in handling hundreds of TCP connections per second. When machines running these implementations were subjected to heavy HTTP traffic, implementation choices like using linked lists instead of hash tables became performance bottlenecks, and were generally corrected by vendor-supplied patches or operating system upgrades. Correcting performance bugs in TCP helps all applications that use TCP, not just webservers.

TCP protocol optimizations - Many TCP implementations do not take advantage of all of the optimizations allowed by the TCP protocol because doing so without application support in a general way is difficult. For example, the TCP protocol allows combining the "FIN" message with the final data segment to reduce the number of packets exchanged. To take advantage of this feature, a TCP implementation must in general know which packet is the final packet before sending it. By allowing the application to provide this information (e.g., through socket options), the TCP implementation can exploit this and other piggy-backing opportunities, thereby reducing the number of messages required per HTTP transaction. The proposed transactional TCP implementation provides a similar approach [3]. All of these optimizations can be enabled through a parametric customization interface, which is inherently safe. Another advantage of this approach is that it avoids network safety issues, because it leaves complex issues such as TCP congestion control in the hands of a trusted and proven TCP implementation.

forking/switching - Early webservers created a new process for each connection, and the associated overhead of forking and context switching limited their performance. Webserver developers responded not by pushing to reduce the cost of forking, but instead by using better designs to avoid forking. Two common options are to have a set of pre-forked processes handle incoming requests, or to use multiple threads within a single process. Even these options have associated overheads, so some servers have moved to using a single, event-driven process. Good application design has essentially solved this problem without the need for OS modification.

filesystem - Filesystem performance is not a problem unique to webservers, nor is it a problem requiring webserver-specific optimizations. The basic problems have been understood for some time, and several groups have implemented successively better solutions in regular, commercial operating systems—the Fast Filesystem (FFS) from BSD [9], the extent-like approach in Sun's UFS [10], and more radical re-designs like SGI's XFS [17]. Changes that broadly improve filesystem performance also improve webserver performance. Due to current HTTP document size distributions, webservers are particularly sensitive to small file performance. Generic filesystem optimizations that improve performance for small files exist. Interestingly, one such filesystem was developed in the context of a project to support webservers through extensible kernels [7]. This confirms our point that experimentation with extensible kernels can lead to generic OS enhancements.

VM/cache control - Applications may know what pages of memory and what files are likely to be used. System calls like `madvise()` exist, and they are inherently safe. Allowing an application to perform file prefetching and cache control has been shown to help performance, but such a system can be implemented using only parameterized interfaces [4] (i.e., by inherently safe customizations). Furthermore, the same mechanisms that allow a webserver to prefetch files after sending a hyperlinked document can be used by applications like compilers that need to read included files.

The three remaining areas where OS performance impacts webservers can also be handled without application-defined kernel extensions. Here, new techniques are required. We briefly discuss the problems before describing our solution.

copying - An efficient webserver using memory-mapped files will still encounter copying when serving regular files, since the networking subsystem must generally copy data from the filesystem into its own buffers. In the case of CGI programs, significantly more copying occurs, since the CGI program communicates data to the server through a socket before the data is sent by the server to the networking subsystem.

double buffering - For each regular client request, the networking subsystem maintains retransmit buffers, duplicating data that already exists in the filesystem cache. This redundancy reduces the effective size of main memory and the filesystem cache.

TCP checksum calculation - Since webservers often send a cached document repeatedly, a possible optimization is the elimination of repeated TCP checksum calculations. If all other copying has been eliminated, TCP checksumming is the last data-touching operation in the server, and may therefore have a significant performance impact.

Our approach to addressing these issues relies on IO-Lite, a *unified I/O buffering and caching system*. IO-Lite allows the safe, efficient, and copy-free sharing of buffers

between applications, the filesystem, the filesystem cache, and the network subsystem. IO-Lite avoids data copying and multiple buffering, and it allows transparent performance optimizations across subsystems, such as caching of precomputed TCP checksums. As a result, IO-Lite provides efficient support for webservers and other I/O intensive applications without any of the "special-casing" the extensible kernel approach requires. From the application writer's standpoint, the benefits come transparently, and within IO-Lite, the benefits come from a unified approach to data transfer, rather than webserver-centric code. A prototype system has been implemented on DEC Alpha workstations running Digital UNIX. More details can be found in Pai et al [13].

Using IO-Lite in a webserver transparently eliminates data copying and double-buffering since applications, filesystem, and networking code can all share the same buffers. More importantly, the same mechanisms apply when using CGI programs, where copying costs can be particularly significant. In comparison, ad-hoc designs that try to integrate the filesystem with the networking system do not benefit CGI programs at all. Our approach preserves "layer transparency," in the sense that no explicit cooperation between the filesystem and networking code is required—each layer operates independently. IO-Lite also enables TCP checksum caching by tagging each buffer with a version number. The combination of the version number and the data address allows the TCP code to cache and re-use checksums. The webserver is not involved with this optimization at all, and once again, "layer transparency" is preserved—the same networking code responsible for the checksums maintains the checksum cache, rather than involving the file system or the application, like the approach used in Cheetah [8].

To demonstrate the benefits of performance-conscious webserver design and IO-Lite, we developed a webserver called Flash, loosely based on thttpd [15]. Without using IO-Lite, Flash already outperforms Harvest [5] and thttpd. When serving small files, Flash beats thttpd by over 200% and Harvest by over 50% (670 connections/sec for Flash versus 391 for Harvest and 214 for thttpd when serving 500 byte files). For large files, Flash beat both by over 100% (211 connections/sec for Flash versus 87 for Harvest and 107 for thttpd when serving 50 kByte files). We conditionally made minor modifications to Flash for IO-Lite support, and the resulting server was dubbed Flash-Lite. For 50 kByte files, Flash-Lite achieved 276 connections/sec, tripling the performance of Harvest, and beating Flash by over 30%. Furthermore, both Flash and Flash-Lite support non-forking CGI programs. When generating 50 kByte responses, CGI applications using IO-Lite ran over twice as fast as their regular counterparts. The server in these tests was a DEC Alpha 200 4/233 workstations connected to a 100 Mbit/s FDDI and a 100 Mbit/s Fast Ethernet network. We also ran the small file test on a 166 MHz Pentium machine running FreeBSD connected to a 100 Mbit/s Fast Ethernet, and we achieved over 1100 connections/sec. More details can be found in Pai et al. [13]

3. Summary

In conclusion, the host of difficult questions raised by extensible kernels suggests that this may be an idea whose time will never come. In experimental systems, extensible kernels may fulfill a meaningful role in allowing relatively rapid prototyping of research ideas that can later be migrated to production kernels. However, most of the effort currently being spent on making extensible kernels "safe" will serve no purpose in this model. Instead, a more fruitful outcome is likely if that same effort were expended on making extensible kernels more usable for researchers. We have shown that the current "killer application" for extensible kernels can be well served (or potentially better served) by a combination of sound application design and general-purpose kernel optimizations, such as those in IO-Lite. In the long term, we feel that the significant issues of complexity, security, and robustness may prevent the commercial adoption of extensible kernel technology, both by operating systems vendors and by users in production environments.

References

[1] T. Berners-Lee, R. Fielding, and H. Hrystyk. Hypertext transfer protocol – HTTP/1.0. Request for Comments 1945, MIT Laboratory for Computer Science (Berners-Lee, Frystyk), UC Irvine (Fielding), May 1996.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[3] R. T. Braden. T/TCP – TCP extensions for transactions. Request for Comments 1644, USC-ISI, 1994.

[4] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *First Symposium on Operating System Design Implementation (OSDI)*, Nov. 1994.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX 1996 Annual Technical Conference*, Jan. 1996.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[7] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX 1997 Annual Technical Conference*, Jan. 1997.

[8] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. In *1996 SIGOPS European Workshop*, Sept. 1996.

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

[10] L. McVoy and S. Kleiman. Extent-like performance from a unix file system. In *Proc. Winter 1991 USENIX Conf.*, pages 33–43, Dallas, TX (USA), 1991. USENIX.

[11] J. C. Mogul, R. F. Rashid, and M. J. Accetta. 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, Nov. 1987.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[13] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. http://www.cs.rice.edu/-vivek/IO-Lite.html.

[14] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[15] J. Poskanzer. thttpd - tiny/turbo/throttling http server. http://www.acme.com/software/thttpd/.

[16] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[17] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *USENIX 1996 Annual Technical Conference*, pages 1–14, San Diego, CA, Jan. 1996.

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