Objects to the rescue!
or
httpd: the next generation operating system

Andrew P. Black
black@crl.dec.com
Cambridge Research Laboratory
Digital Equipment Corporation

Jonathan Walpole
walpole@cse.ogi.edu
Department of
Computer Science and Engineering,
Oregon Graduate Institute

This position paper suggest that object-oriented operating systems may provide the means to meet the ever-growing demands of applications. As an example of a successful OOOS, we cite the http daemon. To support the contention that *httpd* is in fact an operating system, we observe that it implements uniform naming, persistent objects and an invocation meta-protocol, specifies and implements some useful objects, and provides a framework for extensibility.

We also believe that the modularity that is characteristic of OO systems should provide an performance benefit rather than a penalty. Our ongoing work in the Synthetix project at OGI is exploring the possibilities for advanced optimizations is such systems.

---

1. Matching Operating Systems to Application Needs

The call for papers asks how operating systems can meet the ever-growing demands of applications without becoming even larger and more bloated. Ten years ago part of the operating systems community thought we had the answer: the object-oriented operating system.

The idea was that the operating system would become a minimal environment for the support of objects, and that most conventional OS functions would be implemented as objects. The system itself would support low-level naming, object creation and destruction, and object persistence. Functionality such as file systems, memory sub-systems and network protocol stacks would be implemented as collections of objects, each exposing a well defined set of operations (methods) as its interface. The system would also provide a mechanism for invoking these methods.

Extensibility would be one of the hallmarks of such an operating system. New functionality would be supported by adding new objects, or groups of objects. This would have the advantage that the user could select only those functions needed for the application at hand: functionality could be added and removed dynamically.

Or so we thought during the period 1981-1985. It is clear that object-oriented operating systems have not swept all before them. While Macintosh OS and Microsoft Windows do support some degree of dynamic extension, they are not object-oriented in any real sense. Object-oriented operating systems have not displaced Unix or VMS, still less MS-DOS.

Why have OOOS been so unsuccessful? Part of the answer is that compatibility became more important than innovation: whatever else an operating system did, it had to run existing binaries. This is particularly true for MS-DOS and other commodity operating systems. To the Unix community, Mach promised to provide many of the advantages of an object-oriented operating system without really being

one. It therefore stole some of the thunder that would otherwise have been the due of object-oriented operating systems. It wasn't until later that we discovered that the prefix "micro" in the phrase "micro-kernel" referred to functionality and performance, not size and complexity. Mach's image was also damaged by the compatibility bugbear: no matter how novel its features, how elegant its design or how extensible its structure, it could succeed only if its Unix emulation was as good as or better than the native Unix on every platform on which it ran.

Another reason for the lack of success of object-oriented systems is that conventional approaches to implementing modular operating systems appear to introduce too much overhead. The questions of overhead and performance are examined in Section 2.3.

1.1. Is it time to revisit the Object-Oriented Operating System?

This position paper asks whether object-oriented operating systems deserve to be re-examined to see if they can satisfy the needs of modern applications. We look at four issues: the functions that a minimal OOOS kernel should support, the current population of OOOS kernels, what we can learn from them, and performance.

---

2. The Functions of an OOOS Kernel

It is easy to list the functions that an OOOS kernel must support in order to make it possible to build more sophisticated functionality on top of it.

- *Uniform naming*, at a primitive level within a single node, yet providing extensibility to a world-wide internet.
- *Persistent objects*, including their creation, destruction and storage.
- *Delivery of invocation messages* to objects. This includes two things.
  - Definition of a meta-protocol that all objects understand. The meta-protocol describes how an invocation message is received, and how the operation name and the arguments inside it are found.
  - Definition of the interface (protocol) of some common and useful objects.

  It is important to recognize that the messaging metaphor (introduced by Smalltalk) does not correspond to the usual operating system notion of message. Of course, if the objects involved in a particular invocation are physically distributed, then network messages will be necessary. But most invocations are between components on the same machine [2]. The notion of Meta-protocol can refer just as well to the layout of data on the stack as to the layout of data on the wire.
- *Implementations* of some common and useful objects.
- *Extensibility*: the ability to add new objects, and new kinds of objects, to the system, without first taking the system down. Recompiling large parts of the system and re-initializing it is unacceptable; it would be best if no user-visible compilation at all were needed. Dynamic replugging of objects should be a normal activity of an object-oriented system, simply to obtain good performance.
- *Critical mass*: when all of these pieces have been put together, the whole system must do something useful, or else no one will use it. This is the final ingredient of success, which many of the experimental OOOSs, including those that the authors have helped to develop, have lacked. However, ‘‘something useful’’ does not necessarily mean ‘‘something Unix’’.

2.1. The Search for OOOS kernels

If OOOS are indeed the solution to the explosion of demand for functionality, then we should be seeing evidence of their success, not of their extinction. In searching for such evidence, we will take the above list of system characteristics and use it to help us *define* an OOOS, and to answer the question: are there OOOS kernels alive and well and living on the Internet today?

Given this perspective, it should be clear that the World Wide Web is by far the most successful OOOS architecture in use today. In case the reader is not yet convinced that WWW is really an operating system, look at the functions that it provides:

- uniform naming, in the form of URLs;
- persistent object storage, implemented via http daemons and the file systems of their hosts;
- delivery of invocation messages to objects. Http is the meta protocol; the various methods like GET and POST are invocations on object interfaces.
- Multiple implementations of all of the important pieces of the WWW exist; httpd itself implements the most common function (GETting a document from a persistent object).
- Arbitrary user-defined functions can be grafted on, using the Common Gateway Interface (CGI) [7]. However, this is one area in which WWW functionality is less than a traditional Object-oriented system: the CGI interface is essentially a hack designed to allow almost any computation to be initiated by the POST method. A traditional object-oriented system would instead make it simple to build objects that support *new* methods, and for existing client software to invoke them. Maybe this is one area in which the WWW can learn something from Objects?
- The WWW clearly has critical mass: it does something useful. In fact, the WWW may well turn out to be the ''killer application'' of the 1990s. Like Visicalc in the 1980s, it doesn't only do something useful: it does something *so* useful that the application causes people to buy the computer to run it, and thus defines the whole market place.

Another test of object-orientation in operating systems was proposed in 1983 [3]. Given the equivalent of three text files *a*, *b*, and *c*, stored in a system's natural form, how easy is it to construct a fourth file, with the same interface as the other three, whose contents is the dynamic concatenation of *a*, *b*, and *c* – a ''catfile''. Ten years later, catfiles still cannot be constructed on Unix, because of the fundamental dichotomy between files and processes: the process *cat a b c* does not have the same interface as the file *a*. With any http server, catfiles are trivial: a level of indirection allows the server to run the command *cat a b c* and send its output back to the client.

2.2. What can OS designers learn from the WWW?

If the WWW is indeed an example of a successful OOOS, perhaps we as operating system designers can learn some lessons from it that will make our future work more successful. Some of these lessons are debatable, but we believe that the debate should happen: WWW is too big to ignore just because it doesn't fit our pre-conceptions of what a successful operating system should be.

- A single uniform user interface is better than a large number of different special-purpose interfaces. Many of the services available through the WWW – ftp, WAIS, gopher, and news – have been available for years. It is the single consistent user-interface offered by clients like Lynx and Mosaic that make the combination of these services so much greater than the sum of their parts.
- Performance matters, but not nearly as much as is often believed. It is *functionality* that *really* matters: users will pay with performance, provided that they like what they are buying.
- It should be easy to incorporate new applications into the framework. Specifically, we it must be possible to extend the system without writing C language code, and without running a compiler. Experience shows that extensions will happen if they can be constructed in a script language, and if

the interface that they must meet can be described in a couple of pages, not a couple of two-inch-thick volumes.

2.3. Performance and Overhead in OOOS

In Section 1 we observed that conventional approaches to implementing objects appear to introduce too much overhead. Perhaps because of this, the recent trend in micro-kernel-based operating systems has been towards coarse-grained server modules, rather than fine-grained objects. Such coarse-grained modules are typically associated with their own address space and protection domain. Furthermore, each coarse-grained module typically has its own threads. The result of this approach to building modular systems is that object invocation requires thread context switch and address space and protection domain boundary crossings. These are expensive operations, and they are becoming *more* expensive with recent trends in processor design: long pipelines, large register sets and high dependence on cache and TLB performance [1].

Systems such as Chorus [9, 11] have attempted to attack these overheads by allowing coarse-grained servers to be loaded into the same address space and protection domain as the kernel (and each other), hence removing much of the runtime overhead of invoking an object. In a similar vein, other researchers have looked into global address space systems [4] and techniques for software protection [10]. These are steps in the right direction. The Emerald system was an early and successful example of a system in which protection was achieved linguistically, and in which local object invocation was no more expensive than procedure call [5].

The time has come to raise our sights. Our goal has been to reach a point where modularity and performance are orthogonal. That is, we have believed that by implementing modularity in the right way we will be able to achieve the same performance as a monolithic system. We now suggest that modularity ought to be the key to *enhancing* performance. After all, good performance is the result of good optimization. Low-level optimizations, such as replacing an implementation that indexes arrays by one that increments pointers, can often improve performance. However, truly significant performance gains come from choosing the right implementation, such as a hashed lookup rather than a linear search. Thus, good optimization requires that an appropriate implementation is used for each case. The encapsulation provided by object oriented systems should allow us to replug implementations, allowing them to better match their environment. The Synthesis operating system [6] obtains much of its spectacular performance from the dynamic choice of optimal implementations at the module level.

We believe that fine-grain modularity and dynamic replugging will make it possible to perform much more advanced optimizations than are possible in monolithic systems. We are pursuing this approach in the Synthetix project at OGI [8].

---

3. Summary

It is a fact that *httpd* provides a better environment for a wide class of applications than do most of the results of forty years of operating system evolution. We can deny this fact, or we can learn from it. It is certainly too early to give up on object-oriented operating systems on performance grounds, both because performance does not matter as much as is often thought, and because fine-grained objects should provide an opportunity for, rather than an obstacle to, superior performance.

---

4. References

[1] Anderson, T. E., Levy, H. M., Bershad, B. N. and Lazowska, E. D. ‘‘The Interaction of Architecture and Operating System Design’’. *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp.108-120.

[2] Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M. ‘‘Lightweight Remote Procedure Call’’. *Proc. 12th ACM Symp. on Operating Systems Prin.*, Litchfield Park, AZ, December 1989, pp.102-113.

[3] Bhaskar, K. S. ‘‘How object-oriented is your system?’’ *SIGPLAN Notices* **18**, *10* (October 1983), pp.8-11.

[4] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M. and Littlefield, R. J. ‘‘The Amber System: Parallel Programming on a Network of Multiprocessors’’. *Proc. 12th ACM Symp. on Operating Systems Prin.*, Litchfield Park, AZ, December 1989, pp.147-158.

[5] Jul, E., Levy, H., Hutchinson, N. and Black, A. ‘‘Fine-Grained Mobility in the Emerald System’’. *Trans. Computer Systems* **6**, *1* (February 1988), pp.109-133.

[6] Massalin, H. and Pu, C. ‘‘Threads and Input/Output in the Synthesis Kernel’’. *Proc. 12th ACM Symp. on Operating Systems Prin.*, Litchfield Park, AZ, December 1989, pp.191-201.

[7] McCool, R. ‘‘The Common Gateway Interface’’. URL http://hoohoo.ncsa.uiuc.edu/cgi/overview.html, The World Wide Web.

[8] Pu, C. and Walpole, J. ‘‘A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design’’. Tech. Rep. CS/E 93-007, Dept. of Computer Science and Eng., OGI, Portland, OR, April 1993.

[9] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. ‘‘Chorus Distributed Operating Systems’’. *Computing Systems Journal* **1**, *4* (December 1988), pp.305-370.

[10] Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L. ‘‘Efficient Socftware-Based Fault Isolation’’. *Proc. 14th ACM Symp. on Operating Systems Prin.*, December 1993, pp.203-216.

[11] Walpole, J., Inouye, J. and Konuru, R. ‘‘Modularity and Interfaces in Micro-Kernel Design and Implementation: A Case Study of Chorus on the HP PA-RISC’’. *Proc. Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, April 1992.