The Need for Customizable Operating Systems

Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee

Published in Proceedings of the Fourth Workshop on Workstation Operating Systems, pages 165 — 169. IEEE Computer Society Technical Committee on Operating Systems and Applications Environment, IEEE Computer Society Press, October 1993.

(Appears in the proceedings of WWOS'93.)

The Need for Customizable Operating Systems

Gregor Kiczales,[*] John Lamping
Xerox Palo Alto Research Center

Chris Maeda
Carnegie Mellon University

David Keppel, Dylan McNamee
University of Washington

Although modern operating systems provide powerful abstractions to application programs, they often fail to implement those abstractions in a way that provides applications programs, especially specialized application programs, with the best utilization of the physical resources of the computer system[And92].

The operating system community has implicitly recognized this problem by providing mechanisms that give client programmers more access to the physical substrate. The Mach External Pager allows clients to replace the paging mechanism. More recent work, [MA90], [HC92] and [KLVA93], allows client replacement of the paging policy as well. Scheduler activations share the job of thread management between clients and the system. Apertos[Yok92] allows these and other aspects of operating system implementation to be client-controlled. Object-oriented operating systems under development also provide these kinds of control.

We contend that there is a very general issue here, which operating systems have been among the first kinds of software to have to face head-on: some implementation decisions are crucial strategy decisions whose resolution will invariably bias the performance of the resulting implementation. Explicitly recognizing this issue helps to make sense of current trends and suggests new directions to explore. We consider the implications of this issue for operating systems, providing a framework with which to analyze systems such as those mentioned above, and suggesting connections with similar problems in other domains.

1 Mapping Dilemmas

Operating systems are in the business of providing abstractions for services that hide the arbitrary details of raw hardware and that mediate the resource contention among different tasks. This is a laudable goal, and one that intuitively feels like it ought to reduce complexity, so why all the effort to expose more of the implementation issues to clients?

The answer is intuitively accessible, but surprising when given explicit voice:

---

[*] 3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4888; Gregor@parc.xerox.com.

> It isn't possible to hide all implementation issues behind an abstraction barrier because not all of them are mere details, some are instead *crucial strategy issues whose resolution will invariably bias the performance of the resulting implementation.*

We call these implementation issues *mapping dilemmas* because they involve how an abstraction is mapped onto the underlying hardware. That is, they involve what hardware resources constitute the implementation of the abstraction. The mapping dilemmas are those mappings that present substantial strategy issues. We call the decisions on how to resolve them *mapping decisions*. When a service's client performs poorly because the implementation embodies an inappropriate mapping decision, we call it a *mapping conflict*.

Many operating system implementation issues can be crisply stated in terms of mapping dilemmas:

- **Virtual Memory** — The base abstraction of virtual memory is simply a region of memory addresses that can be read or written. The mapping dilemmas are primarily about how to associate virtual addresses with secondary storage, and which of that storage to cache in physical memory at any given time. (This second concern breaks down into individual issues of page replacement policy, page ahead policy etc.)

  A classic example of mapping conflict in this domain is when a client, such as a database system, does a "scan" of one portion of its memory, while doing random access to another portion. A virtual memory implementation based on an LRU replacement policy will perform poorly on the scanned memory.

- **File Systems** — The base abstraction provided by file systems is a system of named files to which data can be read or written in a streaming or random access way. Classic mapping dilemmas include: buffer and buffer pool size, cache management, read ahead, write-through etc. Different applications perform better under different policies.

- **Network Protocols** — The base abstraction provided by a stream-oriented network protocol such as TCP is something like that of a file-system. First, there is a mechanism for getting a connection (essentially naming it), then the connection can be treated as a stream. The mapping dilemmas include such issues as buffer management and scheduling, and detecting and handling lost transmissions.

The concept of mapping dilemmas makes it possible to see that analogous problems crop up in other kinds of systems, including databases and programming languages.

- **Databases** — The abstraction presented by a relational database is just that: a set of relations. There are important mapping decisions having to do with how the relations are implemented, such as what search keys to build indices on.

- **Object-oriented Programming** — The base abstraction is objects with named fields and methods that give rise to behavior. One crucial mapping dilemma is whether to lay the instances out in memory using a dense or a sparse structure. Another is how method lookup should be implemented.

The ways that mapping dilemmas have been addressed in other contexts can inform operating system design, and vica versa, although we don't have space to discuss that here.

2 Consequences for Operating Systems

As the examples above suggest, operating systems have already encountered mapping dilemmas. As hardware gets more sophisticated, with multi-level caches, multi-processors, and multi-level memory hierarchies, the mapping dilemmas become more intricate, because mapping decisions have more consequences.

How can these mapping conflicts be resolved? We want to keep the abstractions that have made operating systems successful, but we want to make sure that the abstractions are implemented in ways that are appropriate for each task. There are several different approaches to resolving mapping dilemmas, and operating system organizations can be characterized in terms of which approaches they take.

The first choice point is whether the mapping decisions are made completely automatically. The traditional approach is to make most decisions automatically, the idea being that if we could make operating systems smart enough, they could automatically choose good mappings for each task. This is the goal that adaptive algorithms, like adaptive schedulers, strive for. But many mapping decisions are so difficult that the level of intelligence required to automatically decide them well is not in sight.

2.1 The user must participate

The alternative to completely automatic resolution of mapping decisions is to have the user (or, in many cases, the application developer) participate in some way in making the mapping decisions. This presents an architectural challenge, because we still want to keep the mapping decisions separate from the abstractions. We don't want to program exclusively at the low level that the mapping dilemmas expose; we want to be able to program primarily at the high level provided by the abstractions while having a say in how they are implemented.

Arguably the simplest approach is to write a special-purpose operating system for each application. Each special-purpose operating system would implement the same abstractions, but would make the mapping decisions most suitable for its intended application. There are two obvious problems with this approach: a new operating system has to be written for each application, and it is hard for different applications to share a single machine.

Rather than start from scratch every time, it makes more sense to have operating systems that can accomodate intervention, customizable operating systems. Anderson[And92] suggested one way of achieving this: "an *application-specific* structure where as much of the operating system as possible is pushed into runtime library routines linked in with each application". We want to understand and evaluate this and other approaches to customizability. Mapping dilemmas provide a framework for the analysis, since they focus on the problems; various architectures can be analyzed in terms of which mapping dilemmas they address.

2.2 Different kinds of mapping dilemmas

We start with the observation that the abstractions of an operating system do two things: They provide some conceptual separation from the arbitrary details of bare hardware, such as paging and disk layout; and they mediate the resource contention among competing demands. This distinction is important both because the different roles present different mapping dilemmas, and because the ways that customization can be achieved are different for the two roles.

The Mach external pager focuses on the first role: providing some conceptual separation from the bare hardware. The external pager lets users write code that determines what secondary storage should hold paged data and how to get data to and from that storage. Notice that the Mach external pager doesn't open up any resource contention issues; the system still decides how many physical pages each task gets, and even which virtual pages will be mapped to physical pages and which won't. The Mach external pager thus opens up one mapping dilemma associated with virtual memory: how virtual memory data should be connected to secondary storage. But it doesn't open up another, the page replacement policy: which pages should be held in physical memory.

The Mach external pager also illustrates a phenomenon that is often associated with openning up a mapping dilemma that spans a conceptual separation from the bare hardware: the utility of an existing interface is extended. In the present example, by letting the user control what the virtual memory maps onto, the open pager lets users use the virtual memory interface — read and write to memory — to access the contents of files, access memory logically shared across a network, or access anything else for which the interface makes sense. Thus, the virtual memory interface goes from being an approximation to physical RAM to being a convenient way to access many kinds of data.

The other role, managing resource contention, logically splits into two sub-roles: managing contention within a protection domain and managing contention among protection domains. Scheduler activations are an example of that split being recognized. Each task is aware of how many processors are allocated to it, and it is responsible for deciding how to utilize them. Contention between tasks is mediated by the operating system, taking advantage of hints from user tasks about whether they can profitably utilize more processors.

An important difference between the two subroles, as illustrated by scheduler activations, is that a protection domain can safely be given complete control over how to deploy the resources allocated to it, while inter-domain resource allocation must still be controlled centrally. This, in turn, affects what mechanisms can be used to express user influence over mapping decisions.

The allocation of intra-task resources can be determined by code run inside the tasks's protection domain, just as code that provides separation from the bare hardware can. By being able to write code, users can express a wide variety of mapping decisions. Of course, it is also important to have facilities that allow simple or common mapping decisions to be expressed easily.

But inter-task resource allocation must be done globally, and protected from untrusted tasks. This means that user tasks can intervene only by indicating to the global allocator which resources are more important to them; no task can have full control. This also means that the range of mapping decisions that can be expressed will be limited by what can be expressed to the global allocator. This points out the importance of splitting intra-task resource management from inter-task resource management so that users can be given maximum control over the resources assigned to them.

3 Conclusion

In conclusion, mapping dilemmas are a principle reason for wanting to open the implementation of operating system functionality. An examination of the mapping dilemmas faced by operating systems suggests areas which should be opened up. Moving responsibility into user tasks is one technique for enabling user control over those issues that can be isolated on a per-task basis. The remaining resource allocation issues must remain under central control to guarantee fairness, but need to be able to accept guidance from the tasks to provide what is most important to each task.

Acknowledgements

Gail Murphy contributed to the development the ideas presented in this paper. We would like to thank Mike Dixon, Paul Dourish, Angie Hinrichs, and Anurag Mendhekar for their comments on earlier drafts.

References

[And92] Thomas E. Anderson. The case of application-specific operating systems. In *3rd Workshop on Workstation Operating Systems, April 1992*, April 1992.

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, 1992.

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 48–64. ACM/SIGPLAN, ACM Press, October 1993. Volume 28, Number 10.

[MA90] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to allow user-level page replacement policies. Technical Report UWCSE 90-09-05, University of Washington, September 1990.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 414–434, October 1992.