Microkernels Should Support Passive Objects

Bryan Ford Jay Lepreau

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA
baford@cs.utah.edu, lepreau@cs.utah.edu

Abstract

We believe that a passive object model, in which the active entities or threads *migrate between passive objects, is more appropriate than an active object model, as the basic structure of a microkernel-based operating system. A passive object model provides enhanced* performance *and* simplicity *because it is more closely matched to the basic nature of microprocessors and the requirements of applications. It also provides more* functionality *by making the* flow of control *between objects a first-class abstraction which can be examined, manipulated, and used to carry information about the operation in progress.*[1]

1 Introduction

A subject of controversy in the object-oriented world is the choice of an *active* or *passive* object model[4]. In the active model, an *object*, or collection of data, has associated with it a private set of *threads*, or active execution contexts, which manipulate its data. To communicate, threads in active objects send *messages* to threads in other objects. In the passive model, objects do not inherently contain threads: only "passive" instructions and data. Threads are separate, first-class entities which exist independently of objects. While a thread is "in" a particular object, it may examine or modify the state of that object, or *migrate* to a different object.

The evolution of modern operating systems toward a microkernel implementation is closely tied to their evolution toward an object-oriented structure. This means that microkernels face the same issues as other object-oriented systems, including the controversy between an active or passive object model. Through our research and experience, we have come to believe that, for maximum performance and functionality, a passive object model is necessary in the basic, bottom-level structure of a microkernel.

We do not make the claim that the passive object model is the best in higher-level parts of the system. Many high-level applications are more naturally implemented in terms of an active object model. Similarly, we do not make the claim that *only* the passive object model should be provided at the lowest level. It may be the case that some uses of a system would highly benefit from a low-level active object model; the two models are not mutually exclusive. However, we claim that *at minimum* a low-level passive object model must be present for maximum performance and functionality.

2 Benefits from Passive Objects

This section describes some of the benefits a passive object model provides over an active model.

Passive objects more naturally support synchronous object invocation, which is the common case.

In practical microkernel-based systems, most inter-object interactions are synchronous. When one object invokes an operation on another, the common behavior is that processing in the first object stops and waits until the operation invoked on the second object is complete. This is purely a practical observation: there are situations in which asynchronous invocation is more useful, but these situations are the exception rather than the rule. The passive object model naturally supports this synchronous flow of control, while the active model requires it to be implemented artificially.

[1]This research was sponsored in part by the Hewlett-Packard Research Grants Program.

Passive objects more closely model the nature of the underlying hardware.

In the context of microkernels, first-class objects are usually implemented as hardware-supported *protection domains* or *address spaces.* Threads, on the other hand, are an abstraction almost always implemented *purely in software.* The hardware only knows about a simple *instruction stream.* A transfer of control between objects in a passive model naturally reduces to the crossing of a protection domain in hardware. Switching to a different thread in an active model as part of inter-object communication has no natural analog in the underlying hardware.

The implementation of inter-object control transfer is simpler and faster with passive objects.

To transfer control between objects in an active model, both the current object *and* the current thread must be changed by the microkernel. In a passive model, only the current object needs to be changed. None of the state related to threads and scheduling needs to be touched. Moving between passive objects is *fundamentally simpler* than moving between active objects, and therefore lends itself to simpler and faster implementations.

The explicit nature of inter-object control transfer makes more optimized implementations possible.

In an active object model, transfer of control between objects during synchronous operations is represented only implicitly by the state and actions of the threads involved; it is not easily visible to the underlying implementation. In an active model, this flow of control is represented explicitly as part of threads, easily visible to and usable by the microkernel.

This permits well-known optimizations to control transfer, such as those described in LRPC[1] and numerous other optimizations in flexibly structured or shared address space systems e.g., Lipto[5], Opal[3][2], FLEX[2], and Mach In-Kernel Servers[8, 6].

Passive objects can be smaller and more lightweight, because they involve less storage overhead.

In an active object model, all objects must "own" a full set of threads and their associated processing and scheduling resources. In a passive model, objects only need to contain their associated code and data, plus a smaller set of system resources needed to allow threads to enter the object and execute its code.

Passive objects more accurately model the requirements of real-time systems.

In the active model, the threads in each object contain scheduling attributes such as execution priority, which are unrelated to the attributes of the threads in other objects. In real-time systems, however, if processing in one object must wait for processing in another object to complete, then the latter processing *must* proceed at the same or a higher execution priority than the former; otherwise priority inversion can result. Satisfying this requirement in an active object model often requires complex *priority inheritance* schemes to be invoked every time control is transferred between objects. In a passive model, execution priority, as part of a thread, naturally flows across object boundaries, obviating the need for priority inheritance mechanisms in this part of the system. The execution priority of an operation is naturally associated with *what is being done* and not *where it is being done*; the passive model more easily supports this.

Passive objects make accurate resource accounting easier.

In a passive object model, resource accounting information can be attached to either threads or objects, whichever is most appropriate in a given situation. This allows resources used by servers on behalf of clients to be charged to the client. An active object model provides no convenient way to do this.

Interruption of operations in progress is more easily implemented with passive objects.

Often, due to asynchronous conditions, it is desired to interrupt an operation invoked on another object. To do this cleanly in an active model, it is not enough merely to wake up the thread in the local object, because the corresponding thread in the server will continue processing the request without any indication that the client no longer desires its completion. If some entity wants to cleanly abort such an operation, it must find the object which was invoked, know how to interact with that object enough to send it a request to abort the operation, and provide it with some kind of identification specifying which operation is to be

[2] Opal claims that threads remain within a single object, but closer examination seems to indicate that the thread actually migrates in the intra-node case.

aborted. This usually proves to be a complex and difficult process. The passive model, on the other hand, provides a channel (the thread) through which standardized requests for interruption can be propagated in a protected manner.

Passive objects are easier to implement and manage in user code.

The implementation of an active object must contain code to create and manage its private set of threads. In practice this turns out to be a complicated task, especially on multiprocessors, because to achieve maximum performance without excessive resource consumption, the number of threads running and waiting for requests from other objects must at all times be carefully balanced to fit the number of processors available. If instead, the object's code is simply executed by threads migrating in from other objects, this balancing occurs automatically.

In a passive model, it is easier for personality servers to control the execution environment of their subjects.

In the case of servers that emulate other operating systems such as Unix or MS-DOS, system calls and exceptions in the emulation environment are typically converted into invocations on the personality server object. In the active model, such an invocation leaves a thread "loose" in the subject domain, which could be woken up by conditions outside the server's control, unless the microkernel and the personality server are very carefully designed to prevent this. In the passive model, once a thread enters the server it is automatically "trapped" by the server; no additional precautions need to be taken other than preventing the subject domain from controlling the server.

3 Common Objections to Passive Objects

In this section we list a number of common objections to the passive object model in the context of microkernels, and why they are not necessarily true.

"An active model is more 'fundamental' than a passive model, because the latter can be implemented in terms of the former."

Either model can be implemented in terms of the other. Implementing passive objects in terms of active objects involves artificially putting to sleep and waking up threads when crossing object boundaries. Implementing active objects in terms of passive objects involves intermediary agents which explicitly maintain asynchronous behavior. (These intermediary agents can be implemented as parts of the calling or called objects, so they do not necessarily imply more objects or more protection-boundary crossing.)

Thus, as far as which is the more "fundamental" model, passive and active objects are on even footing; yet all of the advantages described previously apply easily only to the passive model.

"A passive model provides less protection, because clients must trust servers with their threads."

Migration of a thread to a server object does not necessarily grant the server any rights other than the right to temporarily execute in its scheduling context. With proper design, even this right can be revoked or transferred back into the client object in a way that fully maintains the protection of both the server and the client. Our work on supporting migrating threads on Mach[7] demonstrates how this can be done.

"The migration of threads violates encapsulation. Objects should be independent."

This suggests a vision of self-sufficient islands of computation floating within an abstract sea of nothingness, using only their own resources and relying on nothing else. Unfortunately, reality is different: there must always be some underlying fabric which determines just what it means to be an object, how operations are invoked on it, how it can find and operate on other objects, and what assumptions it can make: in short, the execution environment. A passive object model simply includes as part of the basic fabric the "power to execute," rather than residing only within the object itself.

"It is more difficult to program in a passive model, because all objects must handle internal synchronization issues."

It is true that in an active object model, a simple object can be created requiring no internal synchronization by creating just one thread in the object. However, in a full microkernel implementation of passive objects, it often turns out to be extremely easy to achieve the same effect, such as by creating only one "activation record" on which incoming threads can run, or by maintaining a global lock acquired and released automatically on entry and exit from the object.

"It's easier to implement cross-node object invocation in an active object model, because the underlying hardware is inherently message-based."

In some cases this is true. However, this does not change the fact that the common case of object invocation is synchronous. *At some level* synchronous object invocations must be translated to asynchronous network messages. This argument really just advocates moving this translation out of the microkernel and forcing applications to do it themselves. Considering the fact that most object invocations in a well-tuned distributed system are in fact *local*, where synchronous invocation is more efficient, even this is not a very convincing argument.

"It's not really important whether a microkernel implements passive or active objects as the underlying abstraction, and most conventional operating systems use active objects, so it's best to stay that way for backward compatibility."

First, as we have already stated, object model is in fact very important.

Second, it should be recognized that most "conventional" operating systems are not object-oriented in the first place. What are usually referred to as "objects" in these systems are extremely large, course-grained processes communicating through high-level, heavyweight channels. These traditional channels are typically implemented outside of the microkernel anyway, and therefore are not dependent on the underlying object model supported by the microkernel any more than the rest of the system is.

Finally, we note that from an application's perspective, even ones which directly interface with the microkernel's basic abstractions, the difference in object model is not necessarily highly visible. For the most part, applications merely see passive objects as faster and more flexible versions of active objects. This is demonstrated in [7], in which active objects are replaced with passive objects in an existing system, in a backward-compatible way, with no changes to clients and only minor changes to servers.

4 Conclusion

As a fundamental execution model for microkernels, passive objects provide more functionality, simplicity, and speed than active objects, without giving up protection or other benefits.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993. To appear.

[3] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report UW-CSE-93-04-02, University of Washington Computer Science Department, April 1993.

[4] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[5] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[6] Bryan Ford, Mike Hibler, and Jay Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.

[7] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to use migrating threads. Technical Report UUCS-93-022, University of Utah, August 1993. A portion of this paper will appear in *Proc. of the Winter 1994 USENIX Conference.*

[8] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third USENIX Mach Symposium*, pages 39–55, Santa Fe, NM, April 1993.