Object-Oriented Operating System Design

Vincent F. Russo

Purdue University
Department of Computer Science
West Lafayette, IN 47907 (USA)
Phone: +1 (317) 494-6008
Fax: +1 (317) 494-0739
Email: russo@cs.purdue.edu

1 Introduction

Operating system software should be designed and implemented to allow it to evolve *and remain efficient* in the presence of simple source code maintenance (bug fixes), extensions made for portability, and extensions made to support new features. Not only are requirements of portability, efficiency and extensibility important on their own, but their interactions can require important tradeoffs. For example, if performance modifications increase module interdependencies, portability may be sacrificed for efficiency. Likewise, extensions may impair performance if existing mechanisms must be altered to support or coexist with them.

Using modern programming methodologies, languages and software engineering techniques is the best hope to overcome these problems. The *object-oriented* paradigm is rapidly gaining attention as a useful tool to solve software engineering problems of the type operating system software suffers. Object-oriented programming is claimed to support sharing of common interfaces, sharing of code, incremental extensibility, and the development of reusable and extensible software by allowing functions to be written that can take many different types of objects as arguments.

Although they represent many important ideas, previous operating system design techniques all fail to fully address operating system software engineering problems. My experiences with the *Choices* operating system[Rus90] while at the University of Illinois, and now with the *Renaissance* object-oriented operating system that I'm developing at Purdue, have led me to believe object-oriented techniques are very successful at addressing these problems.

2 The Object-Oriented Paradigm

Before I define an *object-oriented operating system*, I will define what I mean by *object-oriented*. I feel the object-oriented programming paradigm is characterized by four dimensions: *data encapsulation*, *data abstraction*, *inheritance* and *polymorphism*.

Data Encapsulation increases program modularity, maintainability and reliability by localizing data together with functions that operate on them. Data encapsulation in the object-oriented

paradigm is focused on an *object* identified by a name, or reference, to which messages are sent. Sending a message to an object causes a *method* (the code implementing the desired operation) of the object to be invoked.

Data Abstraction is used to identify the types in a program. Data abstraction in object-oriented programming is directed at identifying *classes* to create replicated instances of objects with the same behavior. Type checking in object-oriented languages is implemented by *signature equivalence*. If a given object has the required signature (set of methods), it is appropriate in a given context.

Inheritance is used to allow code sharing (reuse) between classes with similar features and to provide support for the enforcement of common interfaces.

Polymorphism is used to provide code reuse by allowing methods to be written that take objects of many different types as arguments.

Combined, these features provide for *sharing of common interfaces*, *code reuse*, *customization*, and *optimization*. They can help address the problems of operating system *portability*, *maintainability*, and *extensibility*.

Wegner[Weg87] defines an *object-oriented programming language* as an object-based language that supports both classes and inheritance. This definition is in some ways too simple as it does not explicitly mention the need to support polymorphism, which is essential to achieve the benefits of using object-oriented programming. By not using one feature (virtual functions), C++ would still meet this definition of object-oriented. However, it would lack polymorphism since there would be no runtime binding of messages to methods. A better definition is, therefore, that an object-oriented language fully supports programming in the object-oriented paradigm by providing objects as the unit of data encapsulation, classes as the unit of data abstraction, inclusion polymorphism, and some form of sharing of interfaces and method implementations, either by delegation or inheritance.

Examples of object-oriented languages include C++, Smalltalk, Eiffel, Trellis/Owl, SIMULA, CLOS and Modula 3.

Next, I define a software system to be loosely defined as a body of software that is constructed to provide service to other software that relies upon it. A software system must provide an interface to other software that will use the services it provides. Usually this interface takes the form of a small set of subroutines or functions that, when invoked, request services from the system and/or return results. Software systems are distinguished from simple programming libraries by preserving state between invocations of their interface functions, and by interface functions affecting each other's behavior. Operating systems are perhaps the best examples of software systems. Systems providing windowing capabilities on a bit-mapped screen or a database with access/update routines are other examples.

An *object-oriented software system* will be defined by two characteristics. First, an object-oriented software system should be constructed using object-oriented techniques. This means it should be implemented as a dynamic collection of objects, where each object is an instance of some class representing a logical entity the system is modeling. Inheritance and polymorphism should be used to organize the classes and their interrelations. In particular, polymorphism and inheritance should be used to facilitate code sharing, interface sharing, code reuse, and reconfiguration of the system. To simplify using inheritance and polymorphism, an object-oriented system should be

implemented in an object-oriented language. Second, an object-oriented system should provide its services by exposing a select set of constituent objects to external (dynamic) message sends.

A object-oriented system needs to provide a mechanism for applications to obtain references to service objects and to send messages to them. Sending messages to objects replaces the interface functions of a traditionally structured software system. Such a system should also allow additions to the system to be made by inheritance from existing classes and it should support the polymorphism achieved by the late binding of message sends to methods. In short, an object-oriented system is open and extensible within its object-oriented framework.

The Smalltalk environment is perhaps the archetypical object-oriented system. It is a collection of objects that provide a complete programming environment, it is written in an object-oriented language (Smalltalk itself), and it allows new application software written in the environment to have access to any other object already in the environment. The limitation of Smalltalk is that all applications that exist in the environment must be written in Smalltalk.

3 An Object-Oriented Operating Systems

First, and foremost, an *object-oriented operating system* is an object-oriented software system as defined previously. Like any object-oriented system, *all* entities in an object-oriented operating system are represented by objects that are instances of representative classes. This includes encapsulations of: hardware devices, traditional operating system entities such as processes and files, system data structures managing resource allocation and management, system policy modules, and, in particular, low level operating system data structures such as page tables and device control registers. Using an object-oriented language is advantageous, if not essential, to program such a system properly.

Operating systems provide a set of application interface primitives to allow delayed binding of application requests to the operating system functions that implement desired services and to insure integrity of system data and functions. The main feature distinguishing an object-oriented operating system from traditional operating systems is that, like any object-oriented system, it provides its primitives as message sends to system objects. When an application sends a message to a system object, the invocation of the corresponding method must cross any barrier provided to protect the operating system and provide dynamic binding to system services.

An object-oriented operating system needs a naming mechanism to provide applications with references to system objects. One solution is to provide every application with a predefined reference to a *name server* object that maps symbolic names to references. Queries to this object return references to other objects, which in turn provide specific system services. The name server defines the set of all system services available to an application. The classes of these objects, in turn, define the application interface of the operating system for this application. This interface can be extended or reduced dynamically by adding objects to, and removing object from, an application's name server.

4 Advantages of Object-Oriented Operating Systems

Numerous advantage exist for constructing object-oriented operating systems. These include:

- Increased portability.

- Increased code and interface sharing and reuse.
- Support for easy separation of policy from mechanism.
- Optimizations through specialization.
- Support for a structured way to trade portability for efficiency when necessary.
- Support for component testing.
- Support for adaptable interfaces.
- Efficient and localized support for mutual exclusion and synchronization.

With all the benefits of using object-oriented techniques, the natural question to ask is the cost (in terms of efficiency) of using object-oriented programming techniques to construct an operating system. Obviously, the answer to this question depends heavily on the particular object-oriented language chosen. Dynamically typed languages like Smalltalk and CLOS are unlikely to be useful for operating system construction for performance and runtime support reasons. Statically typed object-oriented languages like C++, Eiffel, Modula-3 and Trellis/Owl are probably all reasonable choices.

Another cost associated with an object-oriented operating system is the overhead of accessing system objects across the system/application barrier. Querying name servers may be expensive since it may involve parsing names, but this is only a start up cost since it is performed only to obtain a reference to a system object. Also, this cost is not much different that the similar costs of looking up files or devices through the file system of a conventional operating system. Once the reference is obtained, any performance expense reduces to the cost associated with actually using the reference as the target of a message send. Mechanisms to implement this obviously involve crossing any protection barriers afforded by the operating system. The cost of crossing such barriers is likely to dominate any other messaging cost, and is also incurred in a conventional operating system when invoking an operating system primitive.

The uniformity of accessing data and performing computation using object message sending is an invaluable aid for building distributed systems as well. In conventional distributed systems, data within an application's address space are usually accessed with absolute addresses or by dereferencing pointers to memory, while remote objects are accessed by sending a message to a remote entity. Local data within an application's address space could likewise be accessed by explicit messaging, but this is usually inefficient and harder to program. Object-oriented operating systems easily support the distributed system model since the encapsulation provided by objects allows method invocation to be implemented as local procedure invocations or remote procedure calls (RPC's). All that is needed is to extend object identities to be valid across nodes in a distributed system. Doing this has the advantage of presenting a single model of computation: sending messages to object. Objects within an application access each other this way and objects access objects in other address spaces, or even in address spaces on different computers the same way.

The implementation of such a scheme is similar to the "stubs" used in traditional RPC systems. A reference to an object that is actually remote will, in reality, reference a local *surrogate* object which acts as the stub for the remote object. When a message is sent to a surrogate, its methods marshall the arguments together and send them to a remote surrogate which in turn unmarshalls

them and sends the message to the actual target object. One advantage of such a system is that the inter-surrogate protocol can be specialized for different classes of objects[Sha86]. Generic RPC might work for most cases, but certain surrogates might use a protocol highly optimized for the kinds of methods they are forwarding.

5 Summary

In conclusion, I feel operating systems built using object-oriented techniques:

- can be more maintainable than other systems. In particular that they benefit from:
  - increased modularity, which isolates machine dependencies and increases portability.
  - increased interface reuse, which improves documentation and understanding.
  - increased code reuse, which saves programmer time.
- are more extensible than less structured systems. In particular, their modularity can provide a framework that allows objects to be replaced as the need arises, thus altering operating system policies and mechanisms.
- most importantly, can perform on a par with other systems.

My experiences with both the *Choices* and *Renaissance* systems have show that constructing object-oriented operating systems is a successful technique, both from a software engineering standpoint and an efficiency standpoint. A set of operating system components, maintained as a class hierarchy, provides both an abstract and a practical classification scheme for existing algorithms as well as revealing many new algorithms in its own right. Using a set of operating system algorithms and data structures organized within an object-oriented framework provides useful guidelines to the developer, and reduces any subsequent efforts to retarget or modify the system. It facilitates both customization and optimization of the system.

References

[JLHB87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 62–74, November 1987.

[Rus90] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[Sha86] Marc Shapiro. Strucutre and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[Sha88] Marc Shapiro. The Design of a Distributed Object-Oriented Operating System for Office Applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 168–182, 1987.