Principles of Object-Oriented Operating System Design*

Roy H. Campbell, Gary M. Johnston,
Peter W. Madany, Vincent F. Russo

July 26, 1991

Abstract

This paper presents three principles for the design and construction of object-oriented operating systems. These principles are based on the design of *Choices*, an object-oriented operating system that is implemented on the Encore Multimax shared memory multiprocessor. *Choices* is written entirely in C++, an object-oriented language that supports inheritance through class hierarchies. C++ is compiled, rather than interpreted, so *Choices* executes directly on the hardware without any special run-time support.

The first principle is to *model* the hardware/system software interface by a *framework* of interacting components. The state of the model should reflect the state of the hardware and manipulation of the model should modify the state of the hardware. The second principle is to provide a uniform mechanism for application-to-system and for application-to-application communications. In an object-oriented operating system architecture this should be the invocation of a method on an instance of a system or application object. The third principle is to organize an object-oriented operating system as a member of a *family* of operating systems. This family should be represented as a class hierarchy that forms a *taxonomy* for operating system algorithms and data structures.

We discuss these principles using examples taken from the *Choices* family of operating systems.

Keywords: Object-oriented operating systems, multiprocessor operating systems, distributed operating systems, customizable operating systems, extensible operating systems, object-oriented design, class hierarchies, file systems, stream file systems, virtual memory.

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grant NSG1471.

1 Introduction

Advances in operating system design and architecture require experiments in which theoretical proposals are evaluated by their performance in practical implementations. The outcome of this experimentation often involves observations that can be factored into new theories. In the spirit of experimental evaluation, we have built an object-oriented operating system for a network of shared memory multiprocessors. The system is non-trivial in size and function. It consists of over thirty thousand lines of code and was built by four programmers. It supports three standard file systems, provides virtual memory, distributed virtual memory, multiprocessor parallelism and synchronization, and TCP/IP over Ethernet. It has an object-oriented application interface for parallel applications written in C++, as well as non-object-oriented interfaces for C and FORTRAN. The purpose of this paper is to report three principles that we have applied in our work that govern the construction and architecture of an *object-oriented* operating system.

In our paper, we distinguish between an *object-based* and an *object-oriented* software system. An *object-based* software system uses *encapsulation* to hide data representations, and provides an interface based on *operations* which separates definition and application from implementation. Such a system may include *classes* [19], *modules* [30], or *packages* [23] to define a set of object *instances*. An *object-oriented* software system, on the other hand, includes a *class hierarchy* and uses *inheritance* [12], in which a class defines a template for its instances. Subclasses modify and augment this template. The class introduces definitions of *methods* (or operations) and *instance variables* which are inherited by its subclasses. The methods may be invoked on *all* instances of subclasses of the class.

We also distinguish between an operating system that is constructed as an *object-oriented software system* and one that provides services within an *object-oriented architecture*[1]. The former system has software components that are structured by class and subclass. The latter system provides an environment that supports the dynamic definition of class hierarchies and inheritance for system programs and applications.

The *Choices* operating system [4, 5, 6, 28], built at the University of Illinois, is an object-oriented software system and provides an environment with an object-oriented architecture for system and application programs. The construction of *Choices* has taken about two years and involved many revisions of the code. During this development, we have been led to the following conclusions which we propose as guiding principles for building object-oriented operating systems:

1. A *framework*[2] of interacting components should be built to *model* the hardware/system software interface. The state of the model should reflect the state of the hardware, and manipulation of the model should modify the state of the hardware. In this way,

[1] Interpretive systems like Smalltalk blur our distinctions. Building compiled object-oriented systems using languages like C++ to obtain efficiency makes these distinctions painfully obvious.

[2] The word "framework" has a very specific connotation in object-oriented programming, describing an interaction between instances of subclasses of a set of abstract classes. An example is the Model/View/Controller framework of Smalltalk [12, 15, 17]. It is in this sense that we use it in this paper.

the kernel of an object-oriented operating system can be built as a software system that runs as if it were within an object-oriented hardware environment. The object-oriented modelling approach can be generalized to subsystems and operating system layers. Section 3 describes this in more detail.

2. There should be a *uniform* mechanism for application-to-system and application-to-application communication. In an object-oriented operating system architecture, this mechanism should be the invocation of a method on a system or application object. Like Smalltalk [12] and other object-oriented programming systems, the invocation interface should support the inheritance of methods through a class hierarchy. For example, the mechanism should permit an application to create a subclass of an existing system class in order to substitute a new service for an existing system service.

   The method invocation mechanism should be independent of whether the object

   - is local or on a remote networked machine;
   - resides on disk, in memory, or shares a virtual memory address space with the invoking object; or
   - provides systems services, is a user-defined extension of a system service (an instance of a subclass of the class of the system service), or is an application.

   We discuss this further in section 4.

3. An object-oriented operating system should be organized as a member of a *family* of operating systems. This family should be represented as a class hierarchy that forms a *taxonomy* for operating system algorithms and data structures. The taxonomy represents a theory of the organization of an object-oriented operating system that can be refined in a process of experimentation, evaluation, and revision. It provides an organization for future operating system maintenance, customization, porting, and extensions. We discuss this further in section 5.

This paper is based on the empirical evidence gathered by building an object-oriented operating system for the Encore Multimax multiprocessor system. For brevity, we describe only those features of the operating system that reflect the principles we propose. Further details of the *Choices* operating system have appeared [4, 5, 6, 7, 16, 20, 28] and will appear elsewhere. The source code is available for study under a free license from the University of Illinois.

2 Previous Work

In this section we review the motivations for building new operating systems, the problems of building an operating system, and the benefits of using an object-oriented approach.

2.1 Motivations for Building New Operating Systems

Operating systems evolve to support new hardware architectures and new applications. Evolving operating systems tend to grow and become less modular. Eventually, system developers may suspend this evolution pending a major revision of the operating system code. For example, such a revision occurred in the UNIX operating system when the University of California, Berkeley added paged virtual memory. Occasionally, new operating systems are needed to introduce innovative designs. For example, Multics [9] was introduced to provide time-sharing facilities, and the V [8] system was developed to support message-based systems.

Our motivation for building a new operating system stems from advances in both software and hardware technology. For example:

1. Advances in hardware technology facilitate the concept of distributed virtual memory on networks of shared memory multiprocessors. Programming these machines is more difficult than programming a single shared memory multiprocessor because of the different mechanisms for communication.

2. Network hardware is improving rapidly. For example, the Ultranet^{TM} supports an 800 megabits/second transfer rate. At that speed, communications protocol processing becomes a major bottleneck given conventional distributed operating system architectures. Gigabit/second network hardware may encourage distributed virtual memory, fixed sized messages, and many other innovations and will further increase demands on software.

3. Embedded real-time computer systems require efficient tailorable operating systems with predictable performance. With object-oriented software technology, custom operating systems may be tailored for embedded systems by refining the system from a family described by a class hierarchy.

2.2 Related Research

Organizing an operating system is a difficult task. Many different approaches to structuring operating systems have evolved in response to advances in hardware technology and improvements in software engineering techniques. Operating systems have been organized by hiding machine dependencies using a monolithic kernel [2], by providing a machine independent kernel that hides the details of sharing of a CPU among concurrent processes, by structuring the kernel into layers of processes [11], by providing a functional hierarchy [13], and by representing the mechanisms, services, and abstractions of an operating system as objects [14].

Both the monolithic and kernel approaches are suitable for small operating systems, but do not scale up well because they lack internal structure. The kernel approach helps by separating an operating system into a set of cooperating processes communicating through the kernel. Layering the contents of a kernel is a practical solution to structuring large

operating systems and has been applied to many of today's major operating systems [1, 10, 22, 26].

The major difficulty with building layered operating system kernels is determining the layer in which processes or functions belong and structuring the internals of each layer. Since a layer may only rely on the processes or functions provided by lower layers, careful planning is necessary. For example, in virtual memory systems, the paging mechanisms must use the disk devices drivers. But, the memory that the disk drivers use must be coordinated with the virtual memory management. Such circular dependencies are the most difficult problem in designing the layers of an operating system. The capability-based approach helps to reduce the problems caused by such cyclic dependencies since capabilities can be used to represent arbitrary networks. However, such arbitrary networks lack structure (unless the objects are organized into layers.)

Each approach has merits in terms of simplicity or coping with complexity. Next, we describe how class inheritance is used as a design tool.

2.3 Class Inheritance as a Design Tool

In object-oriented design and programming, an *object* is the basic entity for abstraction. An object is defined as an instance of a *class*. A class defines the interface and encapsulates an implementation for its instances (objects). The class can be thought of as a *template*. The interface defined by a class provides the only means by which other entities in the system can access instances of the class and perform operations upon them.

The interface defined by a class need not always be fully specified. Part or all of it can be taken from other classes. This acquisition of pieces of a class interface from other classes is called *class inheritance*. The classes providing the inherited parts are usually called *parent* classes. The inheriting class is usually referred to as a *sub*-class. Further subclassing leads to the development of a *class hierarchy* in which only the upper levels need be understood in order to infer some of the functionality of the entire system. Class inheritance also allows specification of only the differences between a new class and some existing classes.

All *machine dependencies*, operating system *mechanisms* (e.g., page table management), operating system *policies* (e.g., schedulers) and *design decisions* are *encapsulated* within classes in *Choices*. Design decisions are implemented as a potential set of subclasses. An *abstract* class specifies a general behavior and protocol or signature (the methods that may be used on instances of the class and its concrete subclasses). A *concrete* class refines the implementation of an abstract class. Abstract classes are used in our system design to specify operating system abstractions. Concrete subclasses are used to specify particular versions, policies or mechanisms that implement the abstraction.

In other papers we have discussed the use of this methodology to design process management and exception handling [28], file systems [20], and virtual memory management [6] within *Choices*.

2.4 Structuring an Operating System Using Class Hierarchies

Choices is a study of the applications of class hierarchies and inheritance to the construction of operating systems. In this section, we describe the design methodology permitted by class hierarchies.

We would like to emphasize that the use of *class hierarchies* to structure operating systems is orthogonal to the *layering* of a system. The hierarchies add structure to the internals of a layer or kernel. An object instantiated within one layer may access objects in other layers. Class inheritance allows references to these objects in an organized manner.

The class hierarchical approach also encourages *code reuse* within an operating system layer, between operating system layers, and across systems [18, 20, 21, 28]. We will give examples of these in later sections.

The rest of this paper will focus on further definitions and examples of our design principles for object-oriented operating systems.

3 Abstract Modelling

The first design principle we discovered from our work with *Choices* is that unstructured entities should be hidden by an object-oriented framework through which they can be manipulated. In particular, this requires constructing an *object-oriented model* of the hardware. The model provides an abstraction of entities such as the virtual memory dynamic address translation tables, processor states, DMA, and physical memory. The object-oriented operating system is then implemented using the model. The model is defined in terms of instances of abstract classes. The abstract classes have concrete subclasses that represent specific hardware devices, such as a specific brand and model of memory management unit (MMU). This interface allows operating system algorithms and data structures to be constructed without knowledge of the details of the specific hardware being used. The abstract classes act as place-holders for the concrete subclasses, instances of which are used to implement the system. The model is a framework of interacting class instances because the manipulation of one abstract hardware device may affect other hardware devices. For example, turning the MMU on will impact the addressability of other hardware devices.

3.1 Applications of Modelling in Choices

ProcessContainer. The *Choices* process scheduling framework is a good example of the application of the modelling design technique. An instance of class *Process* encapsulates the state of the hardware required to execute a process. Subclasses of Process include *ApplicationProcess*, *SystemProcess*, and *InterruptProcess*. Each reflects various specializations of class Process. Instances of these subclasses are manipulated by schedulers using the Process class methods.

The abstract class *ProcessContainer* [18, 28] provides the interface for a generic container of Process instances. Instances of subclasses of ProcessContainer provide for the collection,

ordering, scheduling, and execution of processes as shown in Figure 1. The abstract ProcessContainer class defines the methods *add*, *remove*, and *isEmpty*. These methods are redefined by subclasses to impose various disciplines on processes.

The *CPU* subclass of ProcessContainer models the hardware that executes Processes. The CPU subclass is also abstract. In a multiprocessor, instances of subclasses of class CPU represent the processors of the system. Class CPU redefines *add* to dispatch the added process. The *remove* method is redefined to effect process preemption. Subclasses of CPU implement this abstraction for various existing hardware architectures.

The scheduler subclasses of ProcessContainer impose ordering on the processes added to and removed from them. For example, a *FIFOProcessContainer* subclass imposes a first-in-first-out ordering. A *PriorityProcessContainer* imposes a per-process priority ordering.

Exceptions cause processes to be transferred between instances of ProcessContainer subclasses. Exception processing is modelled by adding and removing Processes to and from ProcessContainers.

Subclasses are used to implement specialized policies. For example, the implementation of a semaphore also exploits the ProcessContainer subclasses to provide different policies. The semaphore queue is represented by using an instance of a subclass of ProcessContainer. The particular subclass of ProcessContainer chosen for this instance will affect the behavior of the semaphore. A FIFOProcessContainer provides a standard policy for managing the semaphore queue, but in real-time systems a PriorityProcessContainer might be preferred.

AddressTranslation. Hardware that implements virtual memory must provide dynamic address translation. Examples are page tables, inverted pages tables, and translation lookaside buffers. The *Choices* abstraction for the mechanism is provided by the *AddressTranslation* class. This class provides three primitives to the rest of the virtual memory subsystem. These are *addMapping*, *removeMapping* and *changeProtection*. AddMapping provides a virtual-to-physical address mapping, removeMapping invalidates a mapping, and changeProtection alters the protection of an existing mapping. The abstract AddressTranslation class is all that is needed to implement the rest of a machine independent memory management system. Concrete subclasses of AddressTranslation are created to represent dynamic address translation mechanisms for specific hardware architectures. The instances of particular concrete subclasses of AddressTranslation allow the actual hardware to be completely hidden.

MemoryObject. A final example of a model of hardware that uses abstract classes and concrete subclasses is also derived from the memory management system of *Choices*. The abstract class *MemoryObject* defines access to a logical array of storage units. These units are accessed through a random access *read/write* interface similar to that of a traditional random access file. Concrete subclasses of MemoryObject implement the protocol for disks, disk partitions, and low-level files as shown in Figure 2. Subclasses of MemoryObject also represent various file system, disk paging, and backing store mechanisms. Because of this, virtual memory can be paged on any *Choices* file (including MS-DOS, System V UNIX, and

4.3BSD UNIX), without any algorithmic changes.

4 Uniform Access

Because *Choices* is an object-oriented system (both internally and as an environment), *uniform access* implies that all services should be performed by objects invoking methods on other objects. In *Choices*, method invocation can have one of two characteristics. If both objects involved are in the same virtual address space, the method invocation is called *intra-address-space*. *Choices* is implemented in the C++ programming language [29]. Object method invocation in regular C++ programs falls into this category. If the objects lie in different address spaces, the invocation is called *inter-address-space*. Inter-address-space invocations may involve two objects on the same machine or on different machines. The uniform access principle requires the mechanism of object method invocation to be the same in all three of these cases. In fact, no C++ compiler modifications are necessary to support inter-address-space method invocation in *Choices*. We believe that inter/intra-address-space method invocation is a natural extension of the distributed systems approach of supplying a uniform method (like ports in Accent and Mach [25, 24] or Processes in the V System [8]) for accessing local and remote objects. To support our principle, we first describe our implementation of inter/intra-address-space method invocation. Then we discuss its benefits and application in distributed systems.

4.1 Object Method Invocation

C++ is a statically typed object-oriented language. Every object contains a pointer to a per-class table of the addresses of the entry points of the methods for its class[3]. When a method is invoked on an object, the entry point of the method is looked up in this table at run-time using an index that is determined at compile time. The compiler implements inheritance by copying the entries from the table of the parent class into a table for the subclass. Table entries corresponding to methods that are redefined by the subclass are replaced with the addresses of the entry points of the new method definitions. The principle of uniform access allows a nameserver to return a reference to an object in the same address space, a proxy of an object in another address space, or a reference to a stub object that will perform RPC's [3] to a remote object. A method may be invoked on any of the three without requiring any modification of the application code. Nameservers also support the notion of subclassing and inheritance. A request to a nameserver includes a class defining the the interface that the application will use to access the instance. The actual object returned by the nameserver may be an instance of a subclass of the requested class. Inheritance guarantees the correct object-oriented interpretation of the invocations of superclass methods on an instance of a subclass. This mechanism permits abstract services to be defined by the system or by

[3]In this paper, we refer only to C++ "virtual" member functions. C++ also provides for non-virtual member functions, which are not relevant to this paper.

applications and registered with the nameserver. Subsequently, instances of concrete classes implementing those services may be registered with the nameserver. Applications can be written using the abstract interface and can be bound dynamically to the various instances of the concrete subclasses as desired.

4.2 Inter-Address-Space Method Invocations

The *ObjectProxy* class facilitates inter-address-space method invocation. *Choices* nameservers create and return instances of ObjectProxy in response to a request for an object in another address space. The application uses the ObjectProxy instance to invoke a method on the object in the other address space. Using the standard C++ method invocation described above, the method call is made to the function at the location specified by indexing the ObjectProxy method table. This function maps the method invocation on the target object to the inter-address-space method via whatever communications and mapping mechanisms that are needed. In this way, ObjectProxies act as surrogates for other objects, but this fact is unknown to the client.

The methods of class ObjectProxy trap their invocation to a dynamic method lookup function within the *Choices* kernel. This function implements the actual inter-address-space method invocation using information provided by the nameserver (during the initial request) and the descriptor for the target object (stored in the ObjectProxy instance).

Multiple mechanisms for providing inter-address-space method invocation are possible under *Choices*. Three currently being researched are shared memory, distributed virtual memory, and message passing. Which mechanism to use depends on performance and the relative locations of the two objects involved.

Shared Memory. In *Choices*, an object can be created in an independent virtual memory address space and its lifetime may be independent of any one process. This allows the creation of various forms of servers. In *Choices*, if an application and a server object are resident on the same machine, the kernel lookup function adds the virtual address mappings for the object's data and code to the application's virtual address space for the duration of the call.

Distributed Virtual Memory. In the special case where the objects are both resident on separate machines and both machine are of similar architecture with respect to data representation, a distributed virtual memory mechanism can facilitate method invocation. Two options present themselves in this case. First, the remote object can be made to appear as if it were local to the invoking object's machine by sharing its data (and possibly its code) over the network via paging during the method invocation. This solution works well for small objects or for large objects that are very sparsely accessed. In the case that the remote object is quite large, a second option is to transfer the process from its CPU ProcessContainer on the local machine to a CPU ProcessContainer on the remote machine (i.e., process migration).

Message Passing and RPC A message passing implementation of inter-address-space method invocation to a remote machine is more complicated than the above two cases. Such an implementation does allow distributed systems to have very different processor architectures. In a message passing implementation, the arguments must be copied between the two machines involved. To do this properly, type information must be used to encode the arguments to the method. We are examining the application of existing solutions like RPCs and stubs or wrappers to this problem in Choices. Stubs are responsible for encoding the arguments in messages to the remote machine. A stub at the other end is then responsible for decoding the arguments and finally invoking the method on the remote object.

We feel that the uniformity of accessing objects in any address space via the same mechanism is an invaluable aid in application writing. In conventional distributed systems, objects within an application's address space are usually accessed by dereferencing pointers to memory, while remote objects are accessed by sending a message to a remote process (or a port which represents the remote process). Local objects within an application's address space can be accessed by messaging, but this is usually inefficient. Object method invocation, as implemented in languages like C++, provides the uniform interface with little or no extra cost.

5 Taxonomies

The principle of structuring Choices as a class hierarchy of operating systems algorithms and data structures arose from the effort to design a *family* of operating systems that could be customized for particular hardware architectures and applications. Choices uses abstract classes to define interfaces for its components; these abstract classes are specialized as concrete subclasses in order to provide a particular customization. As we proceeded to build specializations of our operating system components, we discovered that the class hierarchy we created represented a taxonomy[4] of the algorithms and data structures that could be used in customization of specific members of a family. Deliberately building our class hierarchy as a taxonomy would reduce the future effort to adapt the family of systems to include a new customization. The taxonomy represents a theory of the organization of a family of operating systems that can be evaluated experimentally by building new family members. When a new operating system concept is developed that fits into the theory, new subclasses are added to the class hierarchy, and the taxonomy is augmented. If the new concept does not fit into the theory, the class hierarchy must be modified and the taxonomy revised.

The organization within a hierarchy reflects how system designers view the relationships between classes which model similar concepts. The closer two classes are in the hierarchy, the more related should be their corresponding concepts. When distinguishing between major categories of abstract classes, we use *signatures*, i.e., the set of messages or methods to which an object responds. Within each category, however, we arrange subclasses to maximize design and code sharing, since closely related objects are most likely to contain

[4] Webster's dictionary defines a *taxonomy* as the principles covering the classifying of objects.

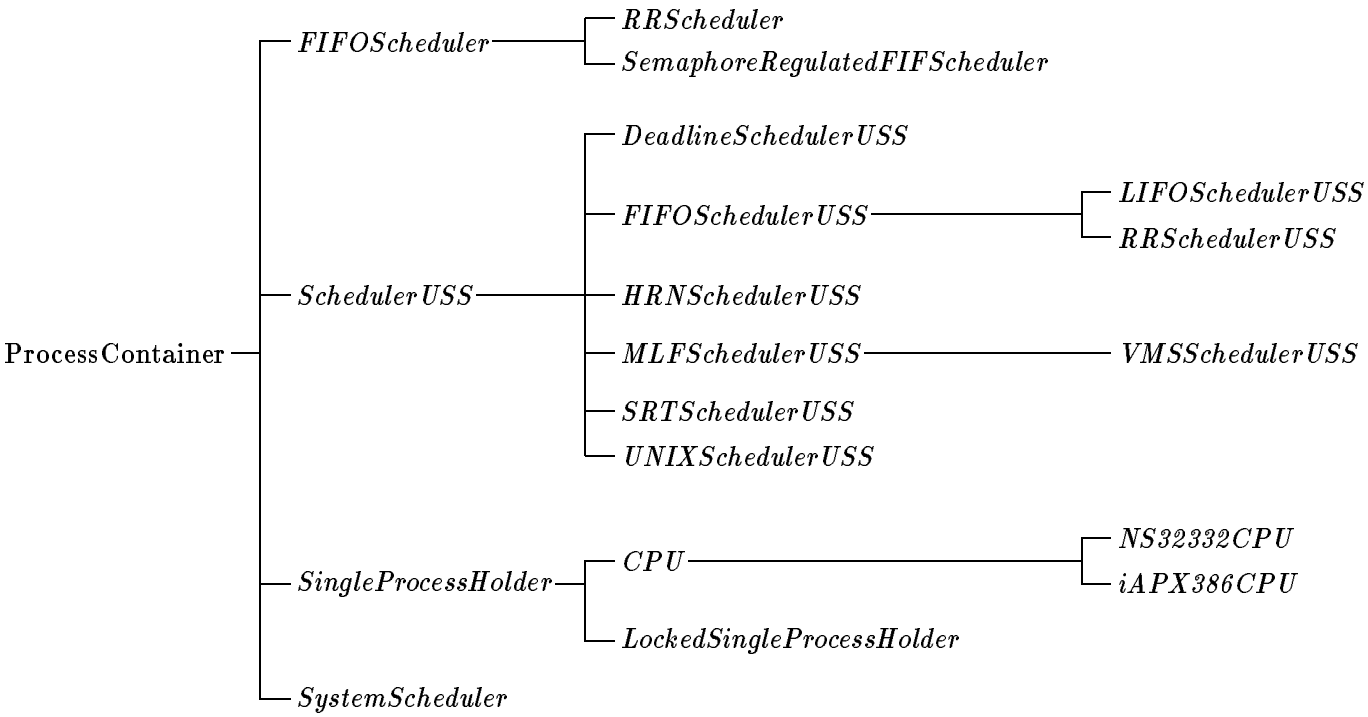

Figure 1: The *Choices* ProcessContainer Hierarchy.

common data structures and algorithms.

Schedulers. *Choices* includes a class hierarchical implementation [18] of the Universal Scheduling System [2, 27]. The abstract class, *SchedulerUSS*, defines the interface for instances of its subclasses. When customizing an individual operating system, a designer can choose one or more schedulers from the concrete classes in the USS hierarchy (see Figure 1) and base these choices on the desired queuing, priority, and preemption policies. Schedulers such as LIFOSchedulerUSS, FIFOSchedulerUSS, and RRSchedulerUSS are based primarily on a queuing mechanism, and are therefore grouped together in the hierarchy. The FIFOSchedulerUSS class can be both instantiated and subclassed. The LIFOSchedulerUSS subclass simply redefines the type of queue to be used, while the RRSchedulerUSS adds the concept of preemption. This is an example of code reuse in the hierarchy since most of the code for LIFOSchedulerUSS and RRSchedulerUSS classes is inherited from the FIFOSchedulerUSS class.

As the scheduler hierarchy grows, it should both widen and deepen, with the superclass-subclass relationship mirroring the theoretical relationship of scheduling algorithms. For example, if one added a shortest-job-first (SJF) scheduler class to the hierarchy, it might become a superclass of the shortest-remaining-time (SRT) class, since the SRT algorithm combines the SJF priority property with the concept of preemption.

Inodes. *Choices* also includes a class hierarchy representing the components of UNIX-like file systems [21]. Both 4.2BSD and System V file systems have been built from components belonging to this hierarchy. Because the major versions of UNIX (BSD and System V) share the concept of an *inode* but specify different data structures for inodes, they are best represented by different subclasses of a common abstract class. The abstract class not only specifies the complete interface or signature for any subclass, but also includes most of the code. The methods of the abstract class rely on a set of data structure manipulation methods that must be defined by the subclasses. This set of methods hides the differences between the data structures of the two specific versions of inodes. The two subclasses simply provide the needed set of methods and inherit the common methods.

Directories. The history of the *Choices* file system directory classes exemplifies the principle of revising a hierarchy when added classes do not fit well into an existing taxonomy. After building our file system hierarchy, we examined it to see if it was general enough to support other stream-oriented file systems; specifically, we tested our hypothesis by trying to add support for MS-DOS. When we added the MS-DOS classes, our perspective changed from a UNIX-oriented view of file systems to a more general stream-oriented view. Thus our class hierarchy required considerable rearrangement. Nevertheless, much of the old hierarchy was reused, including both code for the class definitions and class declarations [20].

We originally viewed directories as subclasses of MemoryObject, an abstract data storage and retrieval object, because they are treated as special files in UNIX. The MS-DOS directories did not fit into the hierarchy. Therefore, we introduced the abstract class, ObjectDictionary, to model logical containers of objects. Concrete examples of ObjectContainer are BSD UNIX directories, System V UNIX directories, and MS-DOS directories. Instead of being characterized by a read/write signature, ObjectDictionaries provide the object naming methods: lookup, add, and remove. They refer to an underlying MemoryObject for the directory's internal data. See Figure 2.

A taxonomy of operating system components, maintained as a class hierarchy, provides both an abstract and a practical classification scheme. Abstract classes define the fundamental functions of components of an operating system. The concrete classes implement these functions in a working member of an operating system family. Using a taxonomy of operating system algorithms and data structures to organize the class hierarchy provides a useful guideline to the developer, reduces any subsequent efforts to port or modify the system, and facilitates customization.

6 Conclusions

We are presently working on the implementation of distributed services using system and application objects, the implementation and optimization of distributed virtual memory, and the incorporation of record-oriented file systems into the class hierarchy.

In addition, *Choices* is being ported to the Intel iPSC/2 80386-based hypercube. This will allow us to augment our class hierarchy in order to take into account operating systems

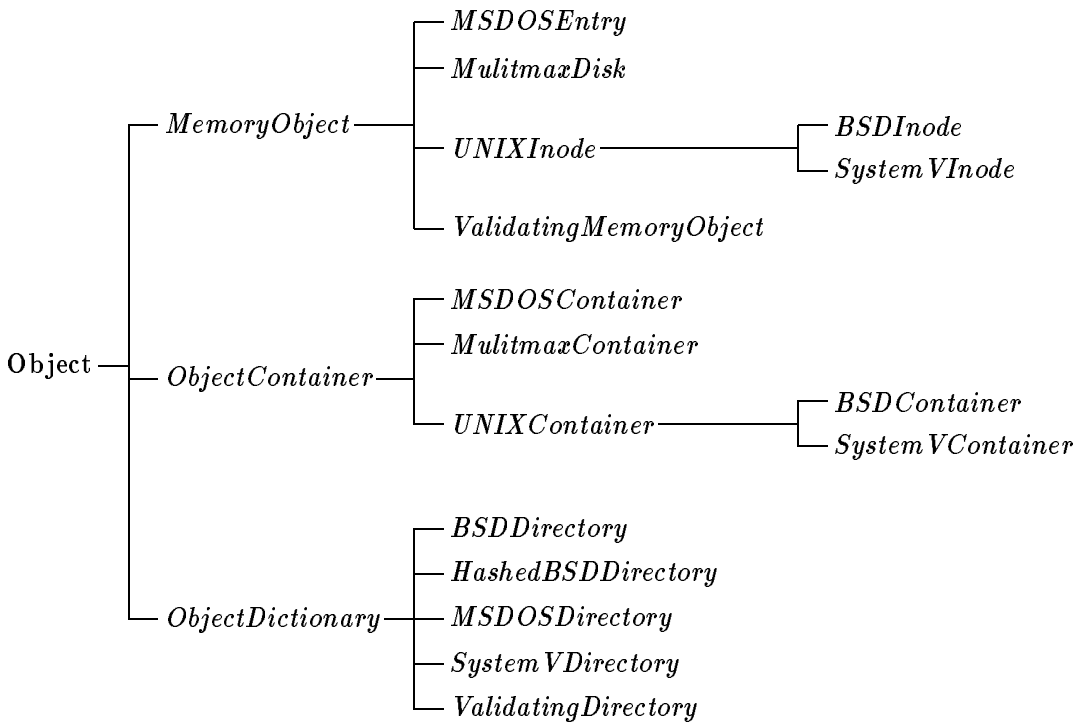

Figure 2: The *Choices* File System Class Hierarchy.

for message-based multiprocessors in addition to shared memory multiprocessors, similar to the way in which the file system hierarchy was augmented by the addition of the MS-DOS file system support.

This paper has described three principles for designing object-oriented operating systems based on modelling, uniform object access, and taxonomy. These principles were derived from our experience of implementing *Choices*. The design and implementation of an operating system is a non-trivial task. We believe that our efforts have benefited from using object-oriented design techniques. It helps us to cope with the complexity of building operating systems for a variety of modern architectures and application areas. We feel that these techniques show promise for building future systems.

References

[1] Mike Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[2] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. Technical Report CSL–83–7, Xerox Corporation, October 1983.

[4] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[5] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[6] Roy Campbell, Vincent Russo, and Gary Johnston. A class hierarchical, object-oriented approach to virtual memory management in multiprocessor operating systems. Technical Report UIUCDCS–R–88–1459, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988. Submitted to ECOOP '89.

[7] Roy H. Campbell and Daniel A. Reed. TAPESTRY: Unifying shared and distributed memory parallel systems. Technical Report UIUCDCS–R–88–1449, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1988. Also Tapestry Technical Report No. TTR88–1.

[8] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[10] Harvey M. Deitel. Case study: VAX. In *Introduction to Operating Systems*, pages 505–533. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[11] Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, pages 341–346, May 1968.

[12] Adele Goldberg and David Robison. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[13] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, pages 266–272, May 1976.

[14] Intel Corporation. *System 432/600 System Reference Manual*, 1981.

[15] Ralph E. Johnson. Model/view/controller, November 1988. Department of Computer Science, University of Illinois at Urbana-Champaign.

[16] Gary Johnston and Roy Campbell. A multiprocessor operating system simulator. In *Proceedings of the USENIX C++ Conference*, pages 169–182, October 1988. Also Technical Report No. UIUCDCS–R–88–1460, Department of Computer Science, University of Illinois at Urbana-Champaign.

[17] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *The Journal of Object-Oriented Programming*, pages 26–49, August 1988.

[18] Douglas E. Leyens. A Choices implementation of the universal scheduling system. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[19] B. Liskov et al. *CLU Reference Manual.* Springer-Verlag, New York, 1981.

[20] Peter W. Madany, Roy Campbell, Vincent Russo, and Douglas E. Leyens. A class hierarchy for building stream-oriented file systems. Department of Computer Science, University of Illinois at Urbana-Champaign, 1988. Submitted to ECOOP '89.

[21] Peter W. Madany and Doublas E. Leyens. A C++ class hierarchy for building UNIX-like file systems. In *Proceedings of the USENIX C++ Conference*, October 1988. Also Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

[22] S. E. Madnick and J. J. Donovan. Virtual Machine/370 (VM/370). In *Operating Systems*, pages 549–563. McGraw-Hill, New York, 1974.

[23] United States Department of Defense. *The Programming Language Ada: Reference Manual.* Springer-Verlag, New York, 1981.

[24] Richard F. Rashid. From RIG to Accent to Mach: The evolution of a network operating system. Technical report, Computer Science Department, Carnegie-Mellon University, 1986.

[25] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1981.

[26] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, pages 365–375, July 1974.

[27] M. Ruschitzka and R. S. Fabry. A unifying approach to scheduling. *Communications of the ACM*, 20(7):469–476, July 1977.

[28] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1988. Also Technical Report No. UIUCDCS–R–88–1415, Department of Computer Science, University of Illinois at Urbana-Champaign.

[29] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[30] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, 1984.