The Applicability of the Object-Oriented Paradigm to the Production of Modern Operating Systems

Wayne Mackrell
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK S7N 5A9
CANADA
e-mail: wlm125@cs.usask.ca

Supervisor: Jean-Paul Tremblay

Abstract

This paper presents a survey of how object-oriented techniques have been used to address the concerns of modern operating system designers. The discussion of this topic includes basic introductions to the concepts which make object-oriented modelling unique, discussions of how object-oriented techniques apply to various issues in modern operating system design, and a brief summary of three operating systems addressing modern concerns which have been implemented using an object-oriented approach. From the evidence presented here, it is apparent that the object-oriented paradigm is appropriate for the efficient and modular design of an operating system.

Introduction

Object-oriented development is quickly becoming the preferred method of software development. The reasons given for this trend usually include discussions about the reusability of object-oriented design and implementation code, its flexibility, its maintainability, and its ability to effectively encapsulate abstract data types in a system. The object-oriented approach is characterized by inheritance, polymorphism, and encapsulation techniques. This paper will discuss the applicability of these techniques to the production of modern operating systems.

Modern operating systems have to deal with concerns at both the hardware level and the application support level which are very different from those which existed only a few years ago. These concerns include object-oriented application support, distributed system support, parallel processing support, and support for a layered, more modular system structure. Modern operating systems are very complex, and the issues associated with their production share this complexity. This paper will attempt to answer whether object-orientation is an effective tool in managing this complexity or not, as well as discussing how it has been used to address certain concerns in modern operating systems.

This paper will first give a brief introduction to the object-oriented paradigm. Three important object-oriented concepts: inheritance, polymorphism, and encapsulation, are discussed in this section. Next, observations obtained from the development of three modern object-oriented operating systems will be presented. This discussion will involve reviewing how object orientation was modelled into the systems, as well as describing the systems themselves. After the system examples, more detailed descriptions of operating system concerns will be presented. The conclusions will address how object-oriented techniques have been used, and whether or not this use has been effective.

The Object-Oriented Paradigm

Introduction

Object-oriented techniques attempt to model conceptualized ideas and real-world entities. An object model of a system consists of a collection of objects interacting to produce the required functionality of the system being described. Intuitively, systems are divided up into components and subsystems. Using approaches which are not object-oriented, these subsystems and components are usually defined by their functionality. As an example, the file system in UNIX is designed to persistent storage of data which is easily accessible by users and the system itself. The file system is implemented with this desired functionality in mind. The data structures are defined, and many procedures are implemented for interacting with these data structures. UNIX, however, does not adequately capture what a file system is at a conceptual level. The data structures are there, but where do they come from? What are they based on? How do they relate to other structures? These aspects of a system are not adequately captured in a design that is not object-oriented.

The focus of an object-oriented approach is to identify the entities in a system. As the system analysis continues, similarities between groups of objects are noted, and simpler, more basic objects are identified which capture these similarities. Through the identification of simple underlying concepts linking many objects, those aspects of the objects can be removed from their design and inherited from the more basic object if inheritance is used properly. In this way, an inheritance hierarchy describing the evolution of objects in the system can be built which captures the fundamental similarities of the objects. However, this is only part of what is required to model a system.

Along with the production of an inheritance hierarchy, developers of object-oriented systems also have to model the interactions between the objects in the system. These interactions are what define the functionality of the system. Many of these relationships can be categorized as client-supplier, in that one object (the supplier) supplies data or a service to another object (the client). Client-supplier relationships are common in object-oriented development, and provide a useful metaphor for modelling the interactions between objects in complex systems.

Many different definitions of the term object-oriented exist in the literature. These definitions range from a mechanism for data encapsulation to an integrated design scheme encompassing the use of inheritance, polymorphism and many other advanced topics. Although considered to be object-oriented by some people, environments not supporting inheritance and polymorphism do not allow some of the most powerful aspects of object-oriented design to be used. These aspects of the object-oriented paradigm will be described in the following sections in more detail.
(Rumbaugh et al. 1991; Nierstrasz 1989)

Encapsulation

When an object-oriented methodology is used to create a system, the design emphasis is shifted from one of coding techniques to one of packaging or modularizing techniques. Encapsulation is the term applied to the packaging of object data structures and methods, which together capture the concepts of an abstract data type. A class in an object-oriented language is an implementation of an abstract data type, and its instantiations in the final system are called objects. Encapsulation consists of bundling all information and routines needed to represent the states and actions of an object together in one package. Internal implementation details are hidden from other objects, and an external interface is used for message passing into and out of the object. This is known as information hiding.

Encapsulation facilitates the independent implementation of different objects in the system. Since implementation details are hidden, the only externally known information about the object is through the external interface it provides. This ensures that changes to an object's implementation do not affect other objects in the system that rely on it. This also ensures that internal data structures will be consistent with the operations an object performs on them without worry of other objects corrupting the data. Without access to the internal details, other objects must be implemented only based upon what services the interface provides. This has the desired effect of reducing the level of coupling present between objects. Coupling has been flagged as a major concern in the maintenance of complex software.
(Rumbaugh et al. 1991; Nierstrasz 1989; Campbell et al. 1989)

Inheritance

Inheritance allows the sharing of common structure between related sets of objects. The main idea behind inheritance is that once a class of objects is defined, another class can inherit all the data structures and methods defined by the first class, and use these structures to form the basis of the new class definition. The most usable object-oriented languages support multiple inheritance, which allows a class to inherit structure from more than one other class. Through the use of multiple inheritance, fundamental concepts contained in many basic classes of objects can be combined in different ways to form new classes with a variety of functionality. When a new class is defined using inheritance, the methods and data structures defined in the parent become defined for the child. Additional data structures and methods can also be defined for the child object class as well to fully define the entity being modelled.

Inheritance provides the object-oriented systems designer with three main advantages. First, the size of methods or routines in a class is reduced. Second, the conceptual clarity of a class base is increased. Third, the idea of inheritance builds upon encapsulation to make the reuse of code more feasible. Although all of these advantages are important, the ease of code reuse is especially helpful in the production of complex systems.

Reuse is one of the most over-used reasons for the switch to object oriented development. Usually when one thinks of code reuse, however, one thinks of reusing code from old defunct systems to make new systems. An object-oriented approach allows for the specifications to be reused within the system as it is being developed. Classes defined in one part of the system may have a functionality that is needed in another part of the system. Inheritance allows new classes to incorporate this functionality, and make any changes to it that are needed to perform the new tasks required.

The inheritance mechanism allows for general ideas about the behaviour of certain object classes to be defined at a highly abstract level. This fundamental idea of inheriting a standard functionality allows developers to model the general structures of entire sets of classes without worrying about the functionality of specific classes in those sets. The ability to access this inherited functionality through the interface of the ancestor class is known as polymorphism.
(Rumbaugh et. al. 1991; Nierstrasz 1989; Campbell et. al. 1989)

Polymorphism

Polymorphism is the term given to describe the ability of an object-oriented system to work with objects at a high abstraction level even though the actual object definition may be several layers down the inheritance structure. As an example of this, consider an object which wants to access a file. The file itself is represented as an object in the operating system. The first object can pass messages to the file object in order to access the functionality provided by its methods and data structures. The file receives the message, performs the required actions, and returns any required data. All of this seems pretty straight-forward. Now consider the situation where the file itself could be an NFS file, a local file, a file stored in memory, a device file (as in Unix) or any of a set of variations. In order to implement code which would be able to handle all these different cases, the routines would be immense. Inheritance, however, allows each of these specific types of files to inherit basic functionality from a parent object, say a file. Polymorphism allows any process accessing these files to operate on the more abstract file object instead of the implementation-specific objects.

Polymorphism allows for flexible implementations of complex systems. This is because the responsibility of discerning which method implementation to invoke is shifted to the data object upon which the operation is being performed. This shift allows the existing code to remain as-is when new classes are added to an inheritance hierarchy, as long as the basic functionality provided by the parent classes is preserved in some form.
(Rumbaugh et al. 1991; Nierstrasz 1989; Campbell et al. 1989)

Examples of Modern Systems

A modern operating system may be faced with hardware environments and application demands which are far different than those faced less than a decade ago. In the modern computing world, the trend is towards a more mobile, distributed, and hardware-independent working environment for the end user. This change has resulted in the development of many operating systems and operating-system add-ons over the last few years to support these added requirements. This section will present three current operating systems which have been built for a variety of purposes. Each subsection will contain a description, a discussion of the specific concerns addressed by the operating system, a brief description of the design methodology used, a discussion about the language requirements, and a set of problems that were encountered in implementing the system. The three operating systems that will be discussed are Choices, Tornado, and Mach.

Choices

Much of the material in this section is based upon information gained by visiting WWW sites pertaining to the Choices operating system, and by reading the papers (Campbell et al. 1989) and (Campbell et al. 1993).

Description Choices is an object-oriented operating system that has been designed and built at the University of Illinois, with the assistance of Sun Microsystems, IBM, and NASA. This operating system was built in an attempt to address the non-uniformity of operating system platforms across different hardware architectures. The design goals for this project were to produce an operating system which would be capable of running on a large variety of hardware platforms while providing a versatile interface for application software development and operating system extension. Inheritance and polymorphism were used extensively in the production of this system.

Considerations Addressed One of the goals for this system was to provide a flexible interface. Specifically, the operating system developers wanted to make it possible to support not one specific implementation of any subsystem, but rather many implementations of subsystems used by application programs. These subsystems include message-passing systems, distributed environments, and parallel processing capabilities. Since most operating systems are specific to one implementation of these various subsystems, applications using these systems are hard to port.

The Choices system has been designed to be hardware-independent. This design was made possible through the use of inheritance, and what the system developers call a framework structure of the operating system. This design technique is described in the following section.

Hardware-independence has allowed the developers to prototype the operating system as an application running on top of an existing operating system. This allowed for easy debugging of the system, and is a common approach in the development of modern systems. Choices was subsequently ported to run native on many different hardware platforms. The Choices operating system has been in use on the SUN SPARC-station, Encore Multimax NS32332, IBM 386 and 486 computers, as well as the Intel Hypercube multicomputer. The operating system is fully capable of running on a uni-processor or multi-processor platform.

Design Methodology The methodology used for the Choices system is based upon frameworks. Frameworks are abstract models of object sets which work together to provide a set of services consistent with an operating subsystem. Frameworks model subsystems as object class sets, and model the interactions between these objects using E-R diagrams and other related techniques. The interactions are defined through constraints, inheritance, and rules of composition. A single subframework in the Choices system defines a single subsystem which has been specialized from the enclosing subframework.

The main framework in the Choices system defines the following notions: process, domain, and memory object. A *process* is a sequentially executed collection of commands with an associated address space, or *domain* composed of *memory objects*. This provides a basis upon which more complicated specializations of the operating system are built. These specializations are called subframeworks. Some examples of the subframeworks implemented are the virtual memory subsystem, the process management subsystem, the file management subsystem, and the message passing subsystem. These subframeworks introduce additional components and constraints as well as further constraining existing components within the subframework. Subframeworks can be refined further, and eventually can be used as a basis for application program implementation.

Each subframework in the system is characterized by a set of attributes which are modelled by data flow diagrams, entity-relationship diagrams, control flow diagrams, path expressions, class hierarchies, and class interfaces. Specializations of the subframework inherit these attributes, and they must be preserved, although they can also be extended. The rules for specializing a subframework are quite simple and are presented below:

- Concrete classes can replace abstract classes.
- Abstract and concrete classes can be added.
- Additional data flows, control flows, entity relations, and path expressions may be added.

Using this methodology, the creators of Choices have been able to model their operating system right down to the hardware interaction level. In order to make the modelling of Choices as hardware independent as possible, the designers needed a way to encapsulate functions performed by hardware in the design, without saying how these functions were to be performed. Using frameworks for specializing system structures into subsystems allowed the designers of Choices to model subsystems without modelling all aspects of their implementation. This allows for a hardware-independent design at most levels of the operating system. Through the use of inheritance and abstract classes, the hardware-specific design can be left until the last stages of specialization when specific methods and objects need to be designed for the host platform.

Programming Language Support The developers of Choices decided to use C++ as the programming language. The reasons behind this decision were that other object-oriented languages either were interpreted and slow, had considerable run-time support, or included pre-defined notions about fundamental parts of the operating system such as processes, synchronization, message passing, and exception handling. The problems with C++ were that it does not support many advanced features of other object-oriented languages. For this reason, the C++ language had to be augmented by Choices developers.

One of the main problems with C++ is that there is no run-time support for object-oriented applications. This problem was addressed by augmenting C++ with the following features (Campbell et al. 1993):

- Garbage collection
- Classes as objects
- Dynamic loading
- Run-time debugging of classes and code

Garbage collection was added to the C++ programming language to facilitate the handling of objects shared across domains. In C++, objects must be explicitly created and destroyed. This necessity to destroy objects can be a burden to the programmer, and hard to implement correctly when dealing with distributed environments. For these reasons, classes in C++ were themselves modelled as objects with increased functionality related to addressing. Under augmented C++, destructors do not need to be called on the objects by the application programmer, and the object addresses are not restricted to the address space of a single machine because of added functionality built into the class object itself. Implementing classes as objects also allows for run-time extensions to the Choices system itself without re-compiling the kernel, as is necessary with most other operating systems. This is a major improvement to the maintenance capabilities for the system. Run-time type checking across domains and run-time debugging are also made possible.

The dynamic loader in Choices allows system programmers to add new services to the kernel at run time. In order to accomplish this, the designers of Choices had to define different kinds of constructors for the C++ objects. C++ statically binds virtual function tables which are the facilities used to execute object methods, including constructors. Dynamically loaded objects therefore cannot be statically bound with the kernel objects already executing. This prevents their methods, including constructors, from being executed by existing code. The key to getting around this limitation is to define a constructor in the object class which is inherited by all objects in the Choices system. Since this constructor is defined in the kernel and inherited by all objects in the system, every object will have the capability of executing this constructor. The class constructor is responsible for ensuring that the C++ constructor is executed.

The dynamic loading feature of Choices programming greatly simplifies the system code. As well, the additional functionality of the system is a welcome improvement in design. The enhancements to the C++ programming language have been at very basic levels, but have greatly improved the functionality accessible to the developer.

Problems Encountered It seems like the main problems encountered by the developers of Choices were those imposed by the programming language C++. This language was chosen because of its efficiency and simplicity. Unfortunately, much of the object-oriented support needed for producing such a complex system is lacking in C++, and extensions had to be made in order to make C++ a decent object-oriented development tool at the operating system level.

Comments on Object-Orientation The designers of the Choices system have advocated the use of an object-oriented philosophy in designing an operating system. They have shown that the performance of object-oriented systems is comparable with conventional systems, and have concluded that the ability to use inheritance and polymorphism was fundamental in producing their flexible operating system.
(Campbell et al. 1993)

Tornado

This section provides information about the Tornado operating system being developed at the University of Toronto. The material in this section is based upon information on the University of Toronto's web site, and the papers (Krieger et al. 1994), (Parsons et al. 1995), and (Unrau et al. 1995). This operating system is designed to run on a large-scale, cache-coherent, shared-memory multiprocessor machine also designed at the University of Toronto. The overall design of the operating system is microkernel based, so that the operating system kernel only provides a minimal amount of services. The servers and daemons which provide much of the high-level functionality we commonly attribute to an operating system run in application or user space. The system is theoretically capable of handling as many as 1024 processors, however, these processors need to grouped together in nodes, each containing only 4 processors. Efficient CPU scheduling can be performed within a node, once a thread has been mapped to that node.

The target applications for Tornado range from computationally complex serial applications to highly parallel scientific calculations. The kind of system support needed for this range of applications varies greatly, and therefore a primary goal of the Tornado designers is to produce a flexible system in which applications could reconfigure resource management facilities in ways which best suit their needs.

Considerations Addressed The Tornado operating system is being designed to address concerns specific to a massively parallel hardware platform. In addition, the designers want to implement a highly flexible interface for application programs to use when interfacing with the operating system. Tornado is intended to provide applications with an operating environment that has predictable performance behaviour. Predictable behaviour allows performance tuning by compiler and application programmers as well as allowing the application to appropriately parameterize its algorithms at run time.

The multiprogramming aspect of the system causes problems with resource usage that can affect performance consistency. As an example, I/O critical applications may expect resources to be configured in a manner most beneficial to that application. Synchronization with the I/O devices is needed to maximize the benefits of the configuration. Interference from other processes, however, can interfere with finely timed I/O operations, and therefore cause high performance loss. The potential for performance loss is even higher in a massively parallel computer due to data distribution across processors, and true concurrency in thread execution. Potential conflicts between concurrently running threads need to be addressed through restraint-constrained scheduling and preloading techniques. These techniques and others are useful for providing user applications with the consistent resource base they need.

The main concerns that need to be addressed for such an operating system are itemized in the following list:

- Handling of large data sets typical of scientific applications.
- Preventing system bottlenecks which reduce effective use of parallel processors.
- Providing a flexible interface for application software.

The development of Tornado using an object-oriented approach has helped to enhance the flexibility of the system. However, it has presented problems with object access across the distributed system. These problems are centred around management of different object representations and cached data across the many processors in a system. These concerns will be discussed more in the next section, and are fully discussed in (Parsons et al. 1995).

Design Methodology Tornado uses a design philosophy called hierarchical clustering which supports information hiding, yet allows localized access to different portions of a distributed object. Externally, objects have a consistent interface to all other objects in the systems with which they may interact. Internally, the object can move around the shared address space, replicate, or distribute data to different nodes in the system as is dictated by performance requirements. This distributed construct is called the object's clustering structure. Each local representation of the object supports the full range of capabilities expected of the object, and appears to clients as an intact object.

Tornado design stresses hierarchical clustering because of concerns about performance. In massively parallel machines, issues such as cache miss percentages become crucial performance metrics of the system. Because of the disparity between the speeds of cache memory and other forms of memory, a cache miss can greatly affect the short-term performance of an operation. On small systems, these concerns are not crucial. However, on massively parallel machines, cache misses on shared data result in fetch operations on that data to caches from other processors. With large numbers of processors, it is easy to see how the system could become bogged down with copying shared data back and forth between the various processor caches. The Tornado operating system is designed to minimize shared data so that the performance degradation associated with cache missing is also minimized.

Programming Language Support Similar to the Choices system, the Tornado operating system is programmed in C++. Again, certain aspects of C++ had to augmented in order to achieve the functionality desired, however, the scope of the changes was not as large as with the Choices system. In the Tornado system, all clustered methods are implemented using C++ virtual functions. Object pointers are not pointers to the local processor or machine, but rather are pointers to entries in an object translation table. Such structures support global access to shared objects across processors and machines.

Problems Encountered As mentioned earlier, one of the main difficulties in designing the Tornado operating system was in devising methods for handling object method execution, data access, and object allocation across multiple address spaces in an efficient and useful manner. The issues with distributed objects are usually dealt with at higher structural levels in the operating system. However, since Tornado is an object-oriented operating system, these issues had to be dealt with at the lower levels in order to effectively handle the objects which make up the system itself. The hierarchical clustering method of object design was the solution to designing efficient systems for handling objects distributed across many address spaces. (Parsons et al. 1995)

Mach

Mach is an operating system produced by the Open Software Foundation, which is based upon the microkernel approach to operating systems development. The Mach system stresses the minimalistic attitude toward kernel development with only basic services being implemented in the kernel, and much of the management capability and application support being delegated to threads running in the user or application space of the system. Much of the information on the object-oriented version of Mach used in this document has been obtained from the OSF site on the World Wide Web. In particular, the document (OSF 1995) provides a lot of detailed information on the MK++ Mach kernel.

The original versions of Mach were based upon the Unix 4.3 BSD kernel code. Essentially, the Mach kernel was originally created by taking the BSD4.3 kernel code and then changing and deleting subsystems one at a time until a new microkernel with thread-managing capabilities was created.

The resulting system was a good, versatile operating system that could support many types of operating system interfaces for a variety of application software domains. Many machines running Mach have application-level servers and daemons that emulate a UNIX environment which is almost unmistakably the same as the environment available under UNIX 4.3 BSD. However, this is not the only interface that Mach can have, and in fact, the micro kernel can be used to emulate many types of operating systems. (Boykin et al. 1993)

OSF fully developed their procedural version of Mach into a distributed operating system capable of handling massively parallel machines in a flexible and efficient manner. Recently, however, OSF has re-implemented Mach using an object-oriented approach. The new version of Mach (MK++) was completed in December, 1995. The MK++ system is scheduled to be functionally complete in mid-March, with the delivery of a complete, tested kernel scheduled for the end of April, 1996.

The object-oriented Mach kernel is based upon six fundamental concepts: tasks, threads, ports, messages, memory objects, and devices. These important concepts are captured as fundamental objects in the kernel, and used to implement the structures which make up the Mach kernel's external interface to application programs. The entire kernel interface is quite small for the object-oriented MK++ Mach kernel. There are only 18 kernel objects accessible by application programs, yet they provide enough functionality so that application programs can easily build file systems and other typical operating system services without having to worry about interacting with the base hardware.

The microkernel idea is very powerful, and seems to be the next step in a natural progression from do-it-all systems to concise, narrowly scoped systems which make up distinct layers of abstraction from the bare hardware. The Mach microkernel is like a virtual hardware platform which is easier to interface with than the actual hardware.

Considerations Addressed The Mach operating system was originally conceived for the purpose of producing a simplistic kernel able to exploit the capabilities present on modern hardware platforms. The idea of minimizing the kernel size involved finding and analyzing the essential services that are required by higher levels, and need to be abstracted from the hardware level. In a minimalistic view, these services or abstractions are all that is needed by a micro kernel to remove application development from hardware concerns. In reality, applications are never removed fully from hardware concerns, as memory shortages, the lack of device support, and computational speed in the CPU can be major limiting factors in the execution of applications on a system.

The other major concern with Mach development was in maintaining compatibility with existing operating system environments. For Mach to be successful, users had to be able to run their programs on it, therefore compatibility became a major issue. Since Unix BSD 4.3 was the dominant operating system in the DARPA group which funded Mach research, Mach's original design was based directly on this version of Unix. Thus, a non-object-oriented, procedural operating system based in C emerged. (Boykin et al. 1993)

A major side-effect of Mach's growth has been the proof that operating system environments such as Unix do not have to be implemented right down to the hardware level in order to achieve efficient execution. In modern Mach systems, the Unix interfaces that applications run in are only applications themselves which are designed to interpret the Unix requests from applications and translate them into service requests that the Mach kernel can handle.

The new design of Mach is not constrained by compatibility concerns at the kernel level, since the previous version has shown that Unix can be implemented on top of Mach to provide compatibility. This shifts the focus to providing a flexible interface to the kernel over which suitable abstractions can be built to provide the functionality of other operating systems. The new Mach kernel had the following goals for its implementation (OSF 1995):

- Exploit parallelism in operating system and user application tasks.
- Support large, potentially sparse address spaces with flexible memory sharing.
- Ensure compatibility with existing environments.
- Ensure portability across platforms.
- Support advanced operating system concepts such as security and real-time applications.

Design Methodology The Mach kernel was designed using a layered approach to operating system development. This approach dictates the implementation of multiple layers, each of which provides services to the layer above and gets services from the layer below (as is quite common in the design of many modern systems). The layered approach lends itself easily to the modelling of inheritance hierarchies. Each layer is modelled as a set of subsystems, each of which is composed of multiple objects. Each layer in the operating system abstracts its view of the hardware given by the layer below it to present a higher level system to the layer above it. This is done through inheritance. Functionality is inherited upwards in the operating system, and new functionality is gained through the combinations of underlying abstractions, and the addition of new information.

Programming Language Support The object-oriented Mach kernel is implemented using C++. The Mach development team found that the limitations of C++ did not present any major barriers, and that its strong control over code generation, and its support for information hiding were very valuable features for producing a high-performance, secure kernel. Early performance results are indicating that because of the ability to model complex performance-enhancing optimizations, the Mach implementation in C++ should surpass the non-object-oriented C implementation in performance.

Comments on Object-Orientation Mach was redesigned using object-oriented principles for the purpose of creating a better structured, more maintainable system. The developers found the object-oriented paradigm to be very helpful in capturing system abstractions and developing a layered, highly modular system. The ability to use inheritance helped considerably in building the layers by providing objects with the basic functionality of each layer. At higher levels of abstraction, inheritance is used to combine the functionality of the lower levels to define new functionality. In the version of Mach which was non-object-oriented, the layering of the system was not as distinct, and the kernel was a mess of procedures, functions, and data structures with no complex relationships modelled in the code. (OSF 1995)

Issues

Modern operating systems have a variety of important issues that they must address. Some of these issues are hardware-oriented, and others involve capabilities that are needed by modern application programs. Hardware concerns can range from interfacing with a wide range of devices to the task of scheduling many processors on a single hardware platform. At the application level, support must be provided for object-oriented systems, concurrent programming, and distributed systems. This type of support is essential for modern application development. Software level support can be implemented in the kernel or outside of the kernel. The movement today is to provide support outside the kernel using application servers themselves. The three modern issues of operating systems design addressed in this paper are: distributed computing, parallel processing, and layered design.

Distributed Computing

Distributed computing involves systems which can operate across network connections, using the resources of many computers in a transparent fashion. Distributed applications are theoretically more reliable than centralized systems because of data and functionality redundancy. Designing support for these types of systems has been an area of major interest recently.

Distributed systems have a high level of complexity because of the large variety of interactions that can take place in these systems. This complexity has been abstracted away by facilities like Remote Procedure Call (RPC), group communication, and distributed shared memory. These abstractions at the operating system level allow for simpler application-level code.

The trend with distributed systems is to implement operating system abstractions for the application layer outside the kernel or micro kernel. One successful implementation based on this approach has been the Distributed Computing Environment (DCE) from OSF. DCE is implemented in the programming language C, but its structure is highly modular and lends itself quite readily to an interpretation using object-oriented concepts. However, implementing an object-oriented support system on a non-object-oriented subsystem has performance costs associated with it that may be prohibitive. Therefore, it is evident that an object oriented approach can capture the design of such a system adequately, but such an implementation requires quick and efficient object support at lower levels to make it feasible. (IBM 1993)

An operating system supporting distributed applications needs to address issues such as file access, object invocation, pointer operations, and thread scheduling across the distributed system. These issues are very similar to concerns in a simple, non-distributed operating system. In non-object-oriented systems, these tasks are performed by having processes in the higher levels of the distributed system interpret specific requests and then delegate processes or threads at lower levels to handle these requests, using network and other interfaces. With an object-oriented approach, the added functionality of the distributed operations could be modelled through inheritance from lower-level non-distributed objects performing the same tasks. This allows distributed support to be moved from the application support level, to a more fundamental system support level.

Inheritance allows application-layer systems to be shielded from the fact that a system may or may not be distributed. An application-layer program can execute as if it is on a dedicated single machine, and leave the interpretation of distributed access to the objects themselves, which may or may not be distributed.

Parallel Processing

Parallel processing describes the use of multi-processor architectures for the execution of software. The issues involved with executing software across multiple CPUs include CPU-cache management, processor scheduling, and other synchronization problems.

These issues introduce a high level of complexity into the CPU scheduler and other low-level operating system objects. At the application level, however, the existence of multiple processors is usually hidden, and application software runs as if only one processor were used (with the exception of concurrent applications whose designers need to be aware that program execution may occur in a truly parallel manner across multiple CPUs, thereby making good concurrent design even more crucial). (Krieger et al. 1994)

Operating systems, by their nature, are not independent from the hardware on which they run. However, in a marketplace where a large number of hardware platforms exist, operating systems which can run on many platforms will be the most successful. Most general-purpose operating systems are built for uni-processor machines, and then are ported to multi-processor machines. Porting an operating system from a uni-processor environment to a multi-processor environment involves modification of low-level routines and structures. This modification of low-level structures is made easier in an object-oriented design where the non-interface features of an object are hidden, and therefore hardware-specific changes can be hidden effectively from higher levels, and reduced in complexity by maintaining a lot of shared functional code between different classes implementing related objects.

Object-oriented techniques have been used successfully in modern operating systems like Tornado and Mach MK++ to effectively model the complexity of multi-processor platforms. Much of the functionality in the objects supporting these platforms is similar to that in objects supporting uni-processor platforms. Using inheritance and polymorphism techniques, the functionality of the multi-processor model can be added to existing uni-processor systems by inheriting the simpler functionality and building upon it to create new objects with the more complex functionality.

Layered Operating System Design

The layered approach to operating system design has been around for a long time. The concept of a layered approach consists of dividing the functionality of an operating system into a hierarchy of systems or layers each providing services to the layer above and using services from the layers below.

In an object-oriented approach to layering, each layer contains sets of objects which define the abstract machine available at that level. Systems at any one layer can view their environment strictly as that defined at the next lowest level, and not worry about the details contained in lower levels. Functionality does not need to be delegated to lower layers as in a non-object-oriented approach, but can be inherited from lower levels and provided as a service at the present layer, perhaps in a more restricted or abstract manner.

Layered implementations have been used successfully in many modern operating systems, including the Mach MK++ kernel. The object-oriented approach facilitates the layering of an operating system. Object-orientation also allows for multiple interpretations of different system layers which can be modelled as objects inheriting different combinations of functionality from lower layers.
(Unrau et al. 1995; Krieger et al. 1994; Graupner, Kalfa, & Scubert 1994; OSF 1995)

Conclusion

Modern operating systems have many complex requirements, both at the hardware level and at the application support level. Modelling the requirements at these levels is difficult using a non-object-oriented approach, and many of the designs end up being based on subsystem hierarchies which can be easily translated into object hierarchies. However, systems designed in this way do not benefit from the focus on data structure enforced by an object-oriented approach, and therefore the code is usually hard to maintain.

The trend in operating system development is to use object-oriented concepts and techniques. Many successes have resulted from object-oriented modelling of operating systems. However, in order to implement these systems, the available language support usually needs to be augmented. The preferred language is currently C++ because of its simplistic nature and its control over the compilation of system code. This language, however, does not support some of the functionality commonly needed in operating systems, and a variety of modifications have been made within the implementation of various operating systems in order to provide the required support. Other languages have also been used for operating systems design, and in some systems, the language is designed concurrently with the operating system itself so that the two become inseparable.

Despite the language difficulties, it appears that the object-oriented methodology is the preferred method for implementing an operating system. The techniques of the paradigm are easily suited to operating system development, and they make the implementation of solutions for many modern requirements much simpler. The consensus seems to be that an object-oriented approach provides better code which is just as efficient as the status quo, and has the extensibility and maintainability properties needed in a large and changing system.

References

Boykin, Joseph; Kirschen, David; Langerman, Alan; and LoVerso Susan. 1993. *Programming under Mach.* Addison-Wesley Publishing Company, Inc., Don Mills, Ontario.

Campbell, Roy; Johnston, Gary; Madany, Peter; and Russo, Vincent. 1989. Principles of Object Oriented Operating System Design. *Technical Report, UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign.*

Campbell, Roy; Islam, Nayeem; Raila, David; and Madany, Peter. 1993. Designing and Implementing Choices: an Object Oriented System in C++. *Communications of the ACM vol. 36, no. 9.* pp 117-126.

Graupner, Sven; Kalfa, Winfried; and Schubert, Fran. 1994. Multi-level Architecture of Object-Oriented Operating Systems. Department of Computer Science Tech Report, Chemnitz University of Technology, Germany.

IBM Canada. 1993. "*Distributed Computing Environment: Understanding the Concepts*". International Business Machines, North York, Ontario.

Krieger, Orran; Gamsa, Ben; Reid, Karen; Lu, Paul; Parsons, Eric; and Stumm Michael. 1994. The Importance of Performance-Oriented Flexibility in System Software for Large-Scale Shared-Memory Multiprocessors. *OOPSLA Workshop on Flexible System Software, October, 1994.*

Nierstrasz, Oscar. 1989. A Survey of Object-Oriented Concepts. *Object-Oriented Concepts, Databases and Applications.* ACM Press. pp 3-21.

OSF. 1995. *MK++ Kernel Executive Summary.* Open Software Foundation.

Parsons, Eric; Gamsa, Ben; Krieger, Orran; and Stumm, Michael. 1995. (De)-Clustering Objects for Multiprocessor System Software. *Appeared at IWOOOS95 (Fourth International Workshop on Object Orientation in Operating Systems), Session 3.*

Unrau, Ron; Krieger, Orran; Gamsa, Ben; and Stumm, Michael. 1995. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing, Vol. 9, Issue 1/2.*

Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; and Lorenson, William. 1991. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey.