Position Paper for the Workshop on Object-Oriented Reflection and Metalevel Architecture at OOPSLA'93

Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach†

Yasuhiko Yokote

Sony Computer Science Laboratory Inc.
Takanawa Muse Building,
3-14-13 Higashi-gotanda, Shinagawa-ku,
Tokyo, 141 JAPAN
e-mail: ykt@csl.sony.co.jp

Abstract

This paper addresses the issues faced when constructing an operating system and its kernel with object-oriented technology. We first propose object/metaobject separation, a means of constructing an object-oriented operating system and its kernel. This method divides the implementing system facilities and applications into two types: objects and metaobjects. This paper presents the concept of object/metaobject separation and discusses why object/metaobject separation is required in terms of limitations in the micro-kernel and object-oriented technologies. We also discuss an example of using object/metaobject separation as implemented in Apertos. This paper then proposes mechanisms which efficiently implement object/metaobject separation. These are characterized by meta-level context management, and are implemented in the Apertos operating system. Meta-level context management is designed to reduce the overhead of control transfer between an object and its metaspace. Here, metaobjects reflectors, *MetaCore*, *Context*, and *Activity* are introduced to represent the metahierarchy of an object's execution. Finally, we present the evaluation results of the Apertos implementation, and discuss the relationship with previous work.

1 Introduction

Recently, object-oriented technology has become popular for the construction of complicated systems, enabling an operating system to again be constructed using that technology. Object-orientation encourages modularization, increases reusability and maintainability, gives users/programmers a single unified perspective of a system, as well as providing other advantages. Example systems are Chorus [Rozier *et al.* 88], Amoeba [Tanenbaum *et al.* 90], Clouds [Spafford 86], and Choices [Campbell *et al.* 91]. Also, micro-kernel technology is widely used for constructing operating systems. A micro-kernel defines minimum functions, on top of which richer system functions are implemented. Systems such as V-kernel [Cheriton 88], Mach [Accetta *et al.* 86], and the systems mentioned earlier all use this technology.

However, recent trends in the computing environment, such as mobile computing and massive-scale distributed computing environments, require a new technology that goes beyond micro-kernel and object-oriented technologies in the construction of an operating system. Although we can characterize this environment using several keywords, this paper focuses its attention on one crucial characteristic, i.e., open-endedness. That is, the system's behavior cannot be predicted from the system configuration time. The increasing scale and complexity of the system also forces users and programmers to reduce their demands on the system, lest it becomes impossible to accurately predict the number of entities, such as workstations, mobile terminals, devices, and activities. Further, due to a high degree of distribution, it is difficult for users and programmers to be freed from their dependence on the environment.

Micro-kernel and object-oriented technologies are limited to addressing the open-endedness property. A micro-kernel provides no policies other than the minimum mechanisms. However, we need a discipline with which policies or system services can be implemented on top of the micro-kernel. Also, since an object is open-ended, it changes its properties during its execution or lifetime. This sometimes requires the extension of the mechanisms provided by a micro-kernel. For instance, if an object provides a realtime constraint, a micro-kernel must provide a mechanism for realtime scheduling. The possibil-

†This work will be presented at International Symposium on Object Technology for Advanced Software (ISOTAS).

ity of the extension of the micro-kernel weakens the advantages of micro-kernel technology.

Object-oriented technology addresses these issues in micro-kernel technology. We, however, encounter difficulties when we support objects in constructing operating system kernels. Even though object-oriented technology offers the advantages of encapsulation (or information hiding) for creating a unified interface between objects, we need a mechanism to break this encapsulation and support objects by operating system kernels in a uniform way.

This paper proposes object/metaobject separation, a means of constructing an object-oriented operating system and its kernel, which divides objects implementing system facilities and applications into two types: base-level objects (or simply objects) and metaobjects. An object is an entity that can be considered as being individual, i.e., it can be shared by another object. A metaobject is a member of a metaspace that provides an object with the optimal execution environment.

The term "meta" is defined in this paper such that:

- it is relative to a "base" (or the base-level of an entity);
- it provides a "base" with an interface and facilities to define the base-level abstraction and semantics that are available to base-level programming; and
- it is an environment in which an object is active.

In this respect a metaspace, in some senses, roughly corresponds to a virtual machine or an optimal operating system for objects. In terms of a virtual machine, a (virtual) instruction set is provided for objects, which defines a software architecture or a software model for object programming. For example, communication protocols between objects, such as synchronous, asynchronous, and realtime protocols, are defined by their metaspaces. In terms of an operating system, an object's execution environment is given by its metaspace, i.e., the metaspace supports an object as a cooperative entity with its environment. For example, if objects need to be scheduled in realtime, their metaspaces provide a realtime scheduler and memory management for that purpose.

In Section 2, we address the concept of object/metaobject separation and discuss why object/metaobject separation is required in terms of the limitations of micro-kernel and object-oriented technologies. We also introduce an example of using object/metaobject separation as implemented in Apertos. Section 3 proposes mechanisms that efficiently implement object/metaobject separation. These mechanisms are characterized by meta-level context management, and are implemented in the Apertos operating system. In Section 4, we present the evaluation results of the Apertos implementation. Section 5 then discusses the relationship with previous work. Finally, Section 6 concludes this paper by discussing the current status of the implementation.

2 Object/Metaobject Separation

As we mentioned in the introductory section, object-oriented technology is again becoming popular in the construction of operating systems. This is because it facilitates the implementation of complicated systems, including operating systems. Before presenting object/metaobject separation, we will discuss the definition of an object. Throughout this paper, we define an object as follows. An object is an entity which can be considered as being an individual. That is, an object is a unit to be shared by another object. Each object has a name (or an identifier) with which others denote that object, i.e., an object has an identity allowing it to be distinguished from other objects.

This basic property creates the following unique advantages. First, it encourages modularization. An object is a unit of protection, which prohibits erroneous and/or malicious access to an object. Actually, the internals of an object are accessed through its public interface as exported to other objects. Second, this property increases the reusability and maintainability. Since the internals of an object or the implementation of an object cannot be seen by other objects, an existing object can be replaced with a new one, if its interface is preserved. This is helped by class hierarchy in object-oriented technology. Finally, it gives users and programmers a single perspective of the system. An object is the only constituent of the system. When we denote a system service, the only entity we can see is the object.

Based on this definition, this section presents object/metaobject separation. Then, we discuss why object/metaobject separation is required, and discuss the limitations of micro-kernel and object-oriented technologies. Also, this section presents applications of object/metaobject separation.

2.1 Proposal of Object/Metaobject Separation

We introduce the notion of "meta" into operating system construction and application programming. That is, objects implementing system facilities and applications are "base" objects (or objects) or "meta" objects (or metaobjects). In detail, object/metaobject separation is defined as follows:

- each object is active in its own execution environment which is given by the group of metaobjects forming its metaspace; and
- a metaobject is an object cooperatively providing an interface and facilities for (the base-level of) objects to define object abstraction and semantics.

This is the principal definition of object/metaobject separation. We extend this below.

Here, consider the construction of a system supporting object-oriented programming. It is difficult to determine an object's granularity during system design. This is because there are three factors; a protection unit, a memory segment which acts as an information container, and an activity which is the thread of control; which should be independently considered and cooperatively designed. UNIX[1] is an example of a system combining these three factors. By introducing threads, the activities can be separated from these three factors.

Recently, the implementation of systems supporting object have shown a trend towards managing these three factors out of the kernel. Amoeba is an example in that its kernel knows nothing about an object. Since an object is a unit of protection, it is protected from others by capabilities managed by server processes. Also, a fine-grained object is protected from others by a programming language. The Amoeba kernel provides a memory segment, but a language system manages that segment for objects. Scheduler activation [Anderson *et al.* 91] and first-class user-level thread [Marsh *et al.* 91] are other examples to handle activities out of the kernel.

With the first definition, an object is free from the internal representation that is usually determined by the underlying operating system. This approach to the above three factors differs from that used in existing systems. Since an object is defined by its metaspace, i.e., an object's identity is given by that metaspace, the policy used to handle these three factors is determined by the metaspace. For example, a metaspace can define objects as an array of structures, but we can regard an element of that array as being an object from outside that object, because the only way to see that object, i.e., to invoke a method of that object, is to use a facility of its metaspace. In this respect, a metaspace shows the objects that are visible from outside the objects.

With the second definition, a metaobject represents a description of an object, i.e., an object's state and a group's state. In this respect, concurrent activities within a metaspace can be introduced to increase the availability of metaobjects up to implementation, and we can define these as concurrent objects [Yonezawa and Tokoro 87]. Also, it is possible for two or more metaspaces to share a metaobject, as a means of keeping common information among those metaspaces. This enables us to reduce the cost of negotiation between metaspaces to solve conflicts between them. By defining a metaobject as an object, the second definition implies that the system can be constructed by the hierarchical structure of objects and metaobjects.

In addition to this definition, we allow an object to change its relationship with its metaspace, i.e., an object can exchange its metaspace with another. This amplifies the advantages of object/metaobject separation. This also facilitates the implementation of object migration. Object migration is when an object travels to another metaspace. Object migration enhances the following capabilities of the system: system extensibility, software upgradability, environment support for mobile computers, environment-dependent system configuration, etc.

2.2 Issues in Micro-kernel and Object-Oriented Structuring

Recently, micro-kernel technology has been widely used to implement operating systems. It is beneficial for the independence of architectural (or hardware) heterogeneity of a system, because it defines the minimum set of system functions, i.e., it can be considered as being standard for inter-operability. It is, however, hard to extend the primitives provided by a micro-kernel when we need to support a new service which cannot be completed without kernel support. An example is a facility for real-time computing. RT-Mach has introduced a realtime facility to the Mach kernel [Tokuda *et al.* 90]. This is a crucial limitation on environments with property open-endedness, because new services that were not expected when the system was configured often emerge.

Micro-kernel technology is also helpful for the modular construction of operating systems. A micro-kernel usually supports a low-level scheduler, virtual memory management, and a mechanism for process communication. Additionally, many system services are implemented as independent modules (or processes in some systems) using this minimum set. It usually defines the basic abstraction of the system, for example in the Mach operating system, processes, threads, objects, and ports are provided by a kernel. In this respect, a kernel divides the system into two layers: one uses the abstraction implemented by the other layer (i.e., it acts as a kernel). Although micro-kernel technology increases the architectural independence of kernel software, it is not the best way of constructing an operating system. It says nothing about realiz-

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

ing independent modules, i.e., no guidelines or rules are provided by that technology. We need no longer be concerned about which services are implemented as modules or which functions are combined into a module.

Object-oriented technology can solve this issue to some degree. That is, it provides a discipline when designing modules, i.e., an object is an independent module, functions are classified by classes, and class hierarchy helps us to reuse existing functions. Also, object-oriented technology addresses the extensibility of the operating system in the sense that objects constituting the system can be dynamically replaced with new ones to support new services. The unified interface for an object helps us to replace an existing object.

However, object-orientation is not acceptable in the following case, because it creates a number of problems. That is, we have to overcome several difficulties when we consider everything as an object. Particularly affected are:

- the debugger, which needs to inspect the internals of an object;
- the object manager, which needs to access meta-data such as the representation of a message and an object's state information to deliver a message to the target and to control object activities; and
- the group manager, which needs to know the dependency between objects.

Since the private methods of an object are permitted to access its internals, we need a special method of exporting these to a debugger. Since meta-data is data that represents an object's state as well as a group's state, it can hardly be maintained by the object itself. If it is separately maintained by an object, it is difficult to keep meta-data consistent with the actual state of concurrently executing objects. Since dependency between objects is dynamically changing and determined according to the degree of global information, we need a way of keeping track of object-object interaction. These difficulties are caused by the object's basic property of being individual.

2.3 Contributions of Object/Metaobject Separation

Due to the issues raised in the previous discussion, we propose object/metaobject separation. Firstly, a metaspace is a collection of metaobjects and is dynamically constructed to be optimal for objects. That is, we can create a new metaspace for a new service. This has advantages over micro-kernel technology in terms of extensibility, and supports the open-ended property.

Secondly, since a metaspace can be a virtual machine devoted to objects, the architectural independence of objects can be achieved in that:

- when an object and its group change their property, we can move them to another metaspace (or a newly created metaspace) to satisfy that change; and
- we can create a new metaspace for a new system service that emerges dynamically.

In this way, the configuration of the system determined at system boot-up can be extended by creating a new metaspace and changing an object's metaspace. This increases the system's reconfigurability and extensibility.

Thirdly, object/metaobject separation can overcome some of difficulties highlighted in the previous subsection. That is, since a metaobject constituting a metaspace represents a description of an object, the internals of an object, the execution state of an object, the dependency between objects, etc. can be accessed through a metaobject. For instance, a debugger is implemented at the meta-level of an object to be examined.

Lastly, in object-oriented technology, there is no distinction between an object and its metaobject for that object. This distinction is crucial when designing complicated operating systems. Object/metaobject separation forces us to be aware of which objects are "meta" of other objects and which objects are running in which metaspace, both during programming and at run time. This clarifies the structure and the configuration of the system. Also, this distinction encourages us to design a new operating system for experimental purposes, which is augmented by the class hierarchy.

2.4 Example of Object/Metaobject Separation

This subsection demonstrates object/metaobject separation in the design of a virtual memory system, implemented in the Apertos operating system. In the design, the memory metahierarchy of the system, which is a metahierarchy dedicated for memory management, has been so designed as to obey the concept of object/metaobject separation. That is, the primary concerns of the virtual memory design are how the system is organized and who is responsible for managing the local storage of an object. Figure 1 shows our answer to these concerns. Here, a shaded rectangle denotes a metaspace, which is represented by a reflector presented later.

At the object-level, we can only handle objects. The internals of an object are protected by the system. At the meta-level of that object, the internals of the object, i.e, its local storage, are represented by several segment metaobjects. A specific property of an object's local storage is given by segment metaobjects. For example, when local

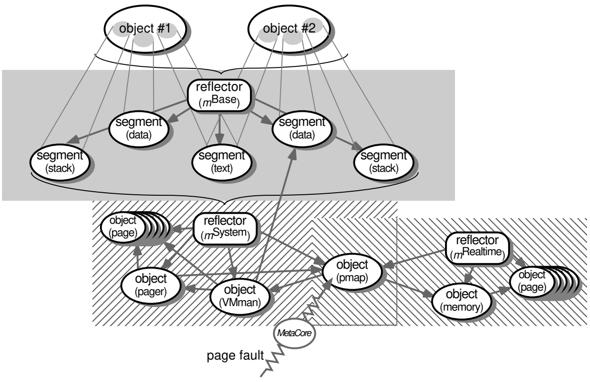

Figure 1: Memory Metahierarchy of the Apertos Virtual Memory System

storage needs to be managed with a specific page replacement algorithm, it is peculiar to an object's segment metaobject. Any area in memory can be managed inside a segment metaobject as a unit for page replacement. The correspondence between the memory chunk size in a segment metaobject and the underlying physical page size is taken by the memory metahierarchy. Also, in Apertos, several policies are provided as classes for constructing the virtual memory system. That is, a segment metaobject is created by a class which implements a property of an object's local storage. Note that, as discussed in [Yokote *et al.* 89], the class hierarchy and metahierarchy is orthogonal in Apertos.

Then, we introduce another metaspace for metaobjects to be objects. In the memory metahierarchy, page metaobjects are the "meta-"metaobjects of a segment metaobject. That is, the local storage of a segment metaobject is represented by page metaobjects at its meta-level. In a similar way, a specific property of a segment's local storage is given by page metaobjects. In Apertos, a page corresponds to a physical memory page, and a segment's page metaobjects are programmed not to move to secondary storage.

According to the definition of object/metaobject separation proposed in Subsection 2.1, an object and its segment metaobject can change their metaspaces to others. In Apertos, this is used to create a dynamically reconfigurable and extensible virtual memory system. The configuration determined at system boot-up can be changed by creating a new metaspace which implements a new property of an object's local storage, and an object changes its metaspace to a new one. This means that an object's local storage is to be managed by a policy that is different to that of the original one. At any level of memory metahierarchy, a metaobject implementing storage management has independence of its metaspace. Hence it can be reconfigurable and extensible.

2.5 Summary

In this paper we claim that the (micro-) kernel of an operating system in an open system cannot provide the minimum set of functions on which system services are constructed, but should provide a means of encouraging object/metaobject separation. Further, object-orientation advocates using an object as an individual with identity and providing object/metaobject separation. Therefore, there are some difficulties in constructing an operating system and its kernel using object-orientation, hence we propose object/metaobject separation in which:

- an object is active in its own execution environment, which we call metaspace;
- a metaobject is an object constituting a metaspace, and cooperatively providing an interface and facilities for (the base-level of) objects to define object abstraction and semantics;
- objects and metaobjects have a relative relationship and are hierarchically structured; and
- an object can replace its metaspace with another.

These are the essential characteristics for addressing the open-endedness property.

3 Implementation in Apertos

An Apertos object is defined as in Section 2, that is, it is an individual and has identity. We make this assumption when implementing object/metaobject separation in Apertos. All metaspaces are constrained by this object definition upon their implementation. This section first discusses the issues of implementing object/metaobject separation in Apertos. We also present the Apertos implementation in terms of the introduction of reflectors, *MetaCore*, and *Context*. We then propose mechanisms that can efficiently implement object/metaobject separation. This is meta-level context management.

3.1 Implementation Issues

Since there are two types of objects in the system, objects and their metaobjects, the following two issues of implementing object/metaobject separation have to be considered:

- how a metaspace is created/constructed; and
- how the relationship between an object and its metaspace is maintained. In particular,
  - when an object interacts with its metaspace, or with a metaobject within that metaspace;
  - when a metaobject interacts with objects;
  - how an object interacts with its metaspace, or with a metaobject within that metaspace; and

– how a metaobject interacts with objects. First, we have to design the metaspace implementation. We have introduced a reflector that is a metaobject that creates a metaspace. We have designed it to be an object that interacts with a metaobject through a reflector representing its metaspace. Thus, an object explicitly invokes a metaobject with the facility provided in its reflector.

Then, we have to design the representation of an object given by its metaspace, i.e., a group of metaobjects. We have clearly separated the three factors discussed in Subsection 2.1. That is, a memory segment for an object as an information container is a metaobject. The storage of an object consists of several memory segment metaobjects in its meta-level, as presented in Subsection 2.4. Since we have designed an object as a concurrent object, a single activity is associated with each object. An activity as a thread of control is represented by a *Context* metaobject. To overcome the protection unit issue, we have introduced reflective object management. The details are discussed in [Yokote *et al.* 91a]. In short, compilers and class systems cooperatively provide optimal protection for an object.

Further, since there is an object running in kernel/system mode, we have to provide a mechanism for changing the object's execution mode. In some implementations, objects running in kernel mode are combined into a single module, usually called a kernel, and are invoked by issuing a system call instruction. Unlike those implementations, we have separated the execution mode (or protection) and the module so that a kernel-mode object can be created outside the kernel.

In these respects, the key is the efficient implementation of control transfer between an object and its metaspace. Table 1 shows the number of system call instructions (`trap` in MC68030 and `syscall` in MIPS R3000) executed for a specified operation. These numbers have been measured for the previous implementation of the Apertos kernel [Yokote 92]. The following subsection proposes the new mechanisms by which an object efficiently passes control to its metaspace.

3.2 The Apertos Implementation

The previous implementation of Apertos is described in [Yokote 92]. We have newly implemented the Apertos kernel based on that experience. The new implementation introduces the following special metaobjects:

- a reflector, representing a metaspace;
- *MetaCore*, located at each processor as a micro-kernel; and
- *Context*, virtualizing the underlying processor for representing an activity.

The following is designed to enable efficient implementation.

The Apertos kernel part is constructed as shown in Figure 2. In the implementation, a metaspace is

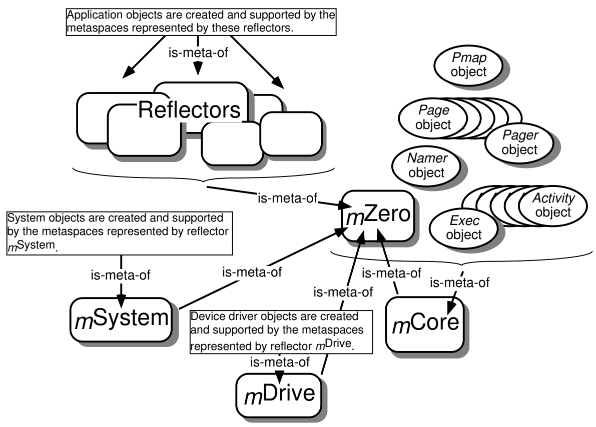

Figure 2: Structure of the Apertos Kernel Part

determined by the references created by a reflector to metaobjects. In the figure, m^{Zero} represents a metaspace consisting of metaobjects *Pager*, *Page*s, *Exec*, *Activity*, and *Namer*, and all of the reflectors in the system are supported by m^{Zero}. To have these metaobjects behave objects, the m^{Core} reflector is introduced. This represents a metaspace consisting of a single metaobject. In the current implementation, m^{Core} assumes the existence of the above metaobjects when Apertos boots, thus avoiding circularity in the memory allocation requests. m^{System} is the metaspace for system objects, including metaobjects implementing the virtual memory system and network protocol handlers. m^{Drive} is the metaspace for device driver objects, which provides facilities for device driver programming.

MetaCore provides the minimum functions that are used for:

- transferring control between an object and its metaspace; and
- handling an external event as the activation of the appropriate *Context*.

These are atomic operations in object/metaobject separation. In the implementation, *MetaCore* has five primitives, as shown in Table 2. Although these primitives are implemented as *MetaCore* methods, *Context* is implemented as a receiver of messages to handle those methods by meta-level context management, as presented in the next subsection.

Before moving the discussion to meta-level context management, we introduce *Context* metaobjects. A *Context* is a meta-level representation of an object's activity and contains enough information to continue the object's execution. *Context*

Table 1: `Trap` Instruction Number upon Execution of a Specified Operation

operation	*# of* `trap`
Method invocation (call/reply round-trip, same metaspace)	4
Method invocation (call/reply round-trip, different metaspaces)	11
Memory segment creation (4KB segment)	44
Object creation (13KB text and 7KB data)	87

Table 2: The *MetaCore* Primitives

primitive	*description*
M	causes the execution of *Context*, designated by the "is-meta-of" link. This causes object execution to stop. Execution is then resumed by primitive R.
R	resumes execution of any *Context* that has been stopped by M.
CActive	returns the reference to *Context* that is currently running.
CBind	associates *Context* with an interrupt. This takes an argument that includes a message to be delivered to an event handling object. When an event is raised, the active *Context* is suspended and its associated *Context* is immediately activated to execute the method of an event handling object.
CUnbind	removes the association made by CBind.

contains no descriptions of virtual memory, but descriptions of an object's thread of control, including its processor registers and execution stacks. *Context* has a significant pointer, which represents the "is-meta-of" link between an object and its metaspace. An object usually has no link to its metaspace. This is maintained by a reflector metaobject. *Context* holds this link as a cache, i.e., two *Contexts*, one for an object's activity and the other for a reflector's activity, are connected by this "is-meta-of" link.

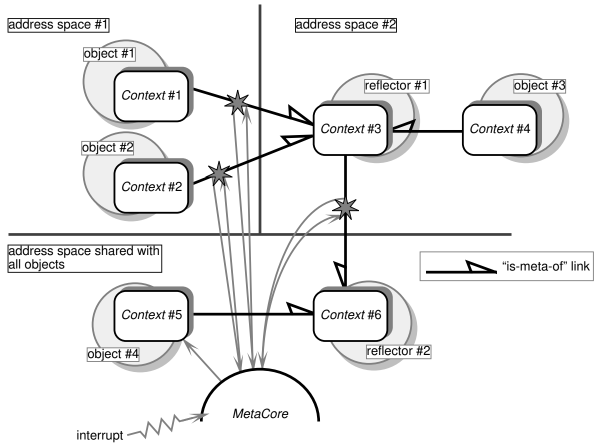

Figure 3: A Possible Configuration of Objects, *Contexts*, and *MetaCore*

3.3 Meta-level Context Management

Since *MetaCore* is the only metaobject constituting the metaspace for *Contexts*, we can create *Contexts* within the address space of *MetaCore*. This, however, causes extra system call instructions, as shown in Table 1. A possible solution to this problem is to allow an object to transfer control to its metaspace without invoking of a *MetaCore* method. That is, *Contexts* are moved out of *MetaCore*, and to be created in any address space. This means that *MetaCore* is free from virtual memory management. Figure 3 shows a possible configuration of objects, their *Contexts*, and *MetaCore*. There are three address spaces in the figure, address spaces #1 and #2, and an address space shared with all other objects. An arrow denotes the "is-meta-of" link between two *Contexts*. For example, a metaspace of object #1 is represented by reflector #1, and this relationship is depicted by an arrow from *Context* #1 to *Context* #3.

Here, when two *Contexts* are in the same address space, there is no need to issue a system call instruction. However, when two *Contexts* are in different address spaces, a system call is still needed to transfer control to the metaspace, as marked with the star in Figure 3. It is complicated to implement a process marked with the star, which is divided into switching an address space to another and locating *Context* representing the activity of the reflector (i.e., the metaspace), because it needs the assistance of the virtual memory system. Also, since *MetaCore* is the only entity accessible from two *Contexts*, it has to provide a primitive to assist the process.

In meta-level context management, we made one assumption to restrict Figure 3. That is, two *Contexts*, one representing the activity of an object's execution and the other representing the activity of a reflector's execution, are assumed to be in the same protection domain. Thus, a system call instruction has to be issued only when the processor's execution mode is changed. Also, this implies that an operation purging the contents of the processor's cache and TLB is independent of switching *Context*. As a result of the above discussion, the control path between communicating two objects can be depicted in Figure 4. In primitive M, exe-

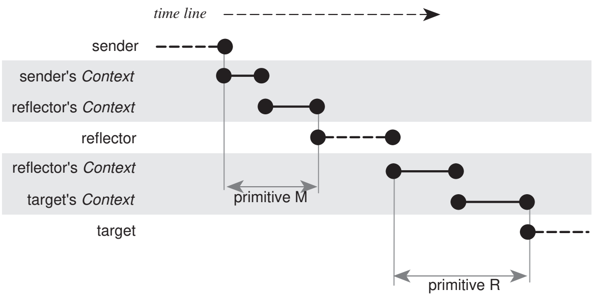

Figure 4: Control Path between Communicating Objects

cution of the sender's *Context* moves to execution of the reflector's *Context*, representing the activity of the sender's metaspace. In primitive R, the reflector's *Context* execution moves to the target. During the processing of the gray area, it is possible to issue a system call instruction to change the processor's execution mode. That is, the latter half of processing primitive M or R may be initiated by a system call instruction. Since this path is still long, we introduce a method cache mechanism, which is denoted by the `cache` entry of *Context*, as in Figure 5. This entry is, in this implementation, maintained by a reflector.

Figure 5 depicts *Context*'s structure in MIPS R3000 implementation, where the following information is maintained.

`status:` is the processor status for this *Context* execution.

`cpu:` stores the contents of the processor registers.

`stack:` denotes a memory segment that is used for an execution stack.

`mode:` is the execution mode. Two modes, user and system, are used.

`mask:` represents the processor's interrupt mask.

`name:` is an object identifier designating to this *Context*.

`meta:` is the "is-meta-of" link to *Context*, associated with the reflector of this object.

`last:` is a reference to the *Context* that most recently activated this *Context*.

`state:` represents the state of *Context* execution.

`entry:` denotes the object's method table to be used by primitives M and R.

`urgent:` denotes a method that is immediately invoked when an external event occurs.

`cache:` denotes a method of handling a method cache that shortens a path between communicating objects.

```
class Context {
protected:
        CPUStatReg      status;
        CPURegisters    cpu;
        Temporary       stack;
        CPUMode         mode;
        longword        mask;
        SID             name;
        CName           meta;
        CName           last;
        mcState         state;
        EntryTable*     entry;
        Entry           urgent;
        MethodCache*    cache;
public: .....
};
```

Figure 5: *Context* Structure

Here, we discuss metahierarchy for an object's execution, depicted in Figure 6. We introduce the

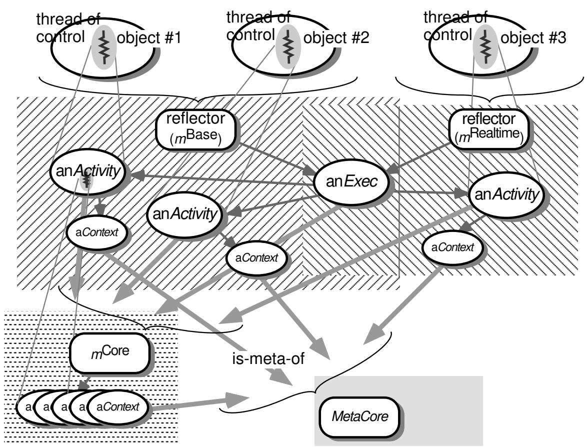

Figure 6: Execution Metahierarchy for an Object's Execution

Exec metaobject. It can be shared by all metaspaces. We also introduce the *Activity* metaobjects, which are a unit of scheduling in *Exec* and are used to manage an object's execution. That is, an object's execution, a thread of control of a method execution, is represented by an *Activity* metaobject at its meta-level. Even though we have already been given it by *Context*, metaobjects *Exec* and *Activity* make it higher, and provide mechanisms for the realtime scheduling for objects. Thus, when a metaspace uses *Exec*, *Activity* is the meta-level representation of an object's execution, rather than *Context*. In the implementation, *Activity* is defined as the data structure in the *Exec* metaobject, but it can be accessed as if it were an object. m^{Core}, the metaspace for *Activity*'s execution, is responsible for this transformation. Also, *Activity*'s execution is represented by metaobject *Context* at its meta-level, i.e., m^{Core}. As described earlier, the metaspace for *Context* is the *MetaCore*.

Table 3 lists the operations provided by *Exec*. Metaobject *Exec* maintains the queue with priority. Users of *Exec* utilize the methods in the table to manipulate *Activity* in the queue. *Exec* itself has no scheduling policy, instead being merely a repository for *Activity*.

4 Evaluation

The results in this section have were obtained using Sony's NEWS workstations, equipped with a 20MHz MIPS R3000 processor, 20MHz R3010 floating point accelerator,16MB of physical memory, and 64KB+64KB caches for data and instructions. When an object and its reflector are assigned to the same protection domain, i.e., they are in the same address space and in the same execution mode, no system call instructions are required. In MIPS R3000 implementation, *MetaCore* is very small, as listed in Table 4. Also, it is designed such that there are no memory allocation requirements inside *MetaCore*.

Table 5 shows the performance of the *MetaCore* primitives, which are invoked without using `syscall`. The numbers in parenthesis are the results with the previous R3000 implementation.

Table 5: Cost of *MetaCore* Primitives (w/o `syscall`)

primitive	*cost (in μsec)*
M	8.7 (21)
R	8.8 (20)

Primitives *CActive*, *CBind*, and *CUnbind* are always used by `syscall` to invoke them. Table 6 shows the use of `syscall`. In the table, ExceptionHandler shows the time interval between the occurrence of an interrupt and the starting of the handler. For this measurement, we used the `break` instruction. The following are the measured results for some operations provided by metaspaces (Table 7). In contrast to the previous implementation, thanks to meta-level context management, these numbers are improved two to five times.

Table 6: Cost of *MetaCore* Primitives (w/ `syscall`)

primitive	*cost (in μsec)*
M	12.1
R	12.6
CActive	4.8
CBind	5.9
CUnbind	5.6
ExceptionHandler	17.1

5 Related Work

Object/metaobject separation is a new way of constructing operating systems. In terms of operating system structuring, many methods have been proposed, including layered structuring, hierarchical structuring, policy/mechanism separation, micro-kernel structuring, object-based structuring, open system structuring, and virtual machine structuring. Detailed discussions of these methods are given in [Yokote *et al.* 91b].

Object/metaobject separation can subsume these structuring methods. In contrast to policy/mechanism separation [Levin *et al.* 75], for example, we can implement an object as a policy module, and a metaobject as a mechanism module. However, this correspondence of object/metaobject separation to policy/mechanism separation does not correctly express the difference. That is, in the sense that a metaobject represents an object's behavior, it should be considered as being a policy module. The major importance is that metaobjects maintain a description of an object, and this is used to provide an optimal execution environment for the object, where there is a possibility of replacing a mechanism if it is not able to provide such an environment.

In contrast to micro-kernel structuring, object/metaobject separation does not divide the system into two layers. It allows us to construct an hierarchical structure of objects and their metaspaces. The ability to change a metaspace facilitates the implementation of object migration, as well as supporting mobile computers, embedded systems, and disconnected operations. Especially, since object/metaobject separation is more tolerant of environmental change than micro-kernel technology, it is advantageous in constructing robot control systems, as in [Brooks 86]. Further, the ability to provide an optimal execution environment for objects is an important capability for massively parallel computers, because the role of each processing element depends on the application. In this way, object/metaobject separation addresses the open-ended property.

Meta-level context management is comparable to the work on user-level thread management done by [Anderson *et al.* 91], [Marsh *et al.* 91], and [Druschel *et al.* 92]. Unlike these systems, however, threads are not managed by objects with kernel

Table 3: The Methods of Metaobject *Exec*

method	*description*
New	creates a new *Activity*.
Delete	destroys an existing *Activity*.
Run	starts the execution of *Activity* that has the highest priority.
Stop	suspends execution of the current *Activity*.
Top	changes the position of *Activity* in the queue.
GetActive	retrieves the identifier of the current *Activity*.
ChangeAttribute	changes the attribute of the specified *Activity*.

Table 4: Size of *MetaCore*

text	initialized data	uninitialized data	total (bytes)
3808	224	3088	7120

support. They are implemented as *Contexts* and are created in a metaspace. The dominant feature of object/metaobject separation is the independence of its objects from their metaspace.

6 Conclusion

Object/metaobject separation makes an object independent of its execution environment (or a metaspace) and encourages an object's metaspace mobility. The existing technologies, such as microkernel and object-oriented ones, are not sufficient to support an operating system and its kernel for an open system. We claim, in this paper, that an operating system should provide a means of encouraging object/metaobject separation.

The Apertos operating system is an example of implementing object/metaobject separation. We propose new mechanisms to efficiently implement object/metaobject separation, because the key to implementation is the interaction between an object and its metaspace. We devise a new technique to enable efficient implementation, i.e., meta-level context management. With this, we can reduce much of the overhead of interaction between an object and its metaspace. In the ideal case, for example, there are no system call instructions and the control path can be shortened between communication objects, maintaining object/metaobject separation.

Current status. The Apertos operating system is being implemented on Sony NEWS workstations, for which two processor architectures, Motorola's MC68030 and MIPS's R3000, are available. Thanks to the MIPS architecture [Kane and Heinrich 92], one notable feature of the implementation is that there is only one local kernel stack. This is also due to meta-level context management and the structure of *MetaCore*.

Apertos is also being implemented on i486-based PC-compatible computers. The kernel part of this implementation is already running and a simple GUI is available. The implementation of Apertos described in this paper (MIPS and i486 implementations) are available to anyone who is interested.

Acknowledgments

I offer my sincere thanks to Prof. Mario Tokoro, the director of the Sony Computer Science Laboratory Inc. He and I have been collaboratively investigating the conceptual design of the Apertos operating system. Also, I would like to thank the members of the Apertos project at Sony Computer Science Laboratory Inc. and Dr. Hideyuki Tokuda of Carnegie Mellon University. Several discussions with these people helped me to design the structure of the system, including the design of Apertos class hierarchy.

References

[Accetta *et al.* 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *USENIX 1986 Summer Conference Proceedings*. USENIX Association, June 1986.

[Anderson *et al.* 91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 95–109, October 1991.

[Brooks 86] Rodney A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE*

Table 7: Cost of Operations Provided by Metaspaces

metaoperation	*cost (in μsec)*
call/reply roundtrip ($m^{\text{Core}} \leftrightarrow m^{\text{Core}}$, w/o method cache)	111
call/reply roundtrip ($m^{\text{Core}} \leftrightarrow m^{\text{Core}}$, w/ method cache)	12.4
call/reply roundtrip ($m^{\text{Base}} \leftrightarrow m^{\text{Base}}$)	193
call/reply roundtrip ($m^{\text{Base}} \leftrightarrow m^{\text{System}}$)	473
call (m^{Zero})	106
reply (m^{Zero})	110

Journal of Robotics and Automation, Vol. RA-2, No. 1, pp.14–23, March 1986.

[Campbell *et al.* 91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 9–15. IEEE Computer Society Press, October 1991.

[Cheriton 88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, Vol. 31, No. 3, pp.314–333, March 1988.

[Druschel *et al.* 92] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 512–520, June 1992.

[Kane and Heinrich 92] Gerry Kane and Joe Heinrich, editors. *MIPS RISC Architecture*. Prentice Hall, 1992.

[Levin *et al.* 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. POLICY/MECHANISM SEPARATION IN HYDRA. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, pp. 132–140. ACM Press, November 1975.

[Marsh *et al.* 91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 110–121, October 1991.

[Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, Vol. 1, No. 4, pp.305–370, Fall 1988.

[Spafford 86] Eugene Howard Spafford. *Kernel Structures for a Distributed Operating System*. PhD thesis, Georgia Institute of Technology, May 1986.

[Tanenbaum *et al.* 90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, Vol. 33, No. 12, pp.46–63, December 1990.

[Tokuda *et al.* 90] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of Mach Workshop*. USENIX Association, October 1990.

[Yokote 92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992*. ACM Press, October 1992. Also appeared in SCSL-TR-92-014 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of ECOOP'89 European Conference on Object-Oriented Programming*, July 1989. Also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 91a] Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Reflective Object Management in the Muse Operating System. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 16–23. IEEE Computer Society Press, October 1991. Also appeared in SCSL-TR-91-009 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 91b] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. *ACM Operating Systems Review*, Vol. 25, No. 2, pp.22–46, April 1991. Also appeared in SCSL-TR-91-002 of Sony Computer Science Laboratory Inc.

[Yonezawa and Tokoro 87] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.