Enhancing Real-time Performance of an Object-Oriented Operating System

by

Robert William Bryce

B.Sc., Brandon University, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. G. C. Shoja, Supervisor (Dept. of Computer Science)

[redacted]

Dr. E. G. Manning (Dept. of Computer Science)

[redacted]

Dr. N. J. Dimopoulos (Dept. of Electrical and Computer Engineering)

[redacted]

Dr. K. F. Li (Dept. of Electrical and Computer Engineering)

Supervisor: Dr. G. C. Shoja

ABSTRACT

This thesis describes mechanisms designed and implemented to enhance the overall, and specifically, real-time performance of a distributed object-oriented operating system called Apertos. Apertos employs a meta-hierarchy, defined by the relationship between an object and its supporting environment, which is intended to support various objects with different requirements, such as real-time support and persistence. As such a system grows, satisfying different object requirements via the meta-hierarchy with its related communication overheads becomes orthogonal to achieving real-time response performance. To address this performance penalty and to improve real-time support, we introduce preemptive scheduling and hierarchical scheduling as solutions. Preemptive scheduling improves the stability of the system. Hierarchical scheduling establishes a more flexible system in terms of scheduling, and improves communication performance by a large factor over the original scheme applied.

Examiners

[redacted]

Dr. G. C. Shoja, Supervisor (Dept. of Computer Science)

[redacted]

Dr. E. G. Manning, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. N. J. Dimopoulos, Outside Member (Dept. of Electrical and Computer. Engineering)

[redacted]

Dr. K. F. Li, External Examiner (Dept. of Electrical and Computer Engineering)

Table of Contents

ABSTRACT ii
Table of Contents iii
List of Tables vii
List of Figures viii
Acknowledgments ix

Chapter 1 Introduction 1
1.1 Object-Oriented Paradigm 1
1.1.1 Class 2
1.1.2 Object 3
1.1.3 Metaobject 4
1.2 Object-Oriented Operating Systems 6
1.3 Apertos 7
1.4 Objectives 8
1.5 The Remainder of this Thesis 9

Chapter 2 Background and Related Work 10
2.1 Overview of Apertos 10
2.1.1 Goals of Apertos 10
2.1.2 Communication Model 11
2.2 Other Operating Systems Which Directly Support Objects 13
2.2.1 Chorus 13
2.2.2 Amoeba 15
2.2.3 Clouds 16
2.2.4 PM Operating System 17
2.2.5 Spring 18
2.2.6 Choices 19
2.2.7 Other Object-Oriented Operating Systems 19
2.3 Comparison 23

Chapter 3 Current Implementation of Apertos 24
3.1 Active Objects 24
3.2 Activity and Context Classes 26
3.3 Reflectors 28

3.3.1 A Reflector Template: MCommon ... 28
3.3.2 User Reflectors: MMiniMeta and MCOOP ... 28
3.3.3 Device Driver Reflector: MDrive ... 30
3.3.4 System Reflectors: MCore and MZero ... 30
3.4 MCore Metaobjects ... 31
3.4.1 Segment, SystemSegment, and Mapper ... 31
3.4.2 Namer ... 32
3.4.3 Exec ... 32
3.4.4 Promise ... 33
3.4.5 Idle ... 33
3.5 MetaCore ... 33
3.6 Communication between Active Objects ... 34
3.7 Object Interfaces ... 35

Chapter 4 Scheduling ... 41
4.1 Present Scheduling Policies ... 41
4.2 Preemptive Scheduling ... 43
4.2.1 Providing Support for Preemptive Scheduling ... 44
4.2.2 Integrating Objects into the System ... 46
4.2.3 Preemptive Scheduling Policy ... 47
4.2.4 Preemption in Previous Versions of Apertos ... 47
4.2.5 Other Modifications Required ... 48
4.3 Integrating Real-time Constraints ... 50
4.3.1 Real-time Scheduling ... 50
4.3.2 Reflection in Real-time Scheduling ... 50
4.4 Hierarchical Scheduling ... 52
4.4.1 Hierarchical Scheduling as a Solution ... 52
4.4.2 Fitting the Apertos Paradigm ... 52
4.4.3 Previous Work ... 53
4.4.4 The Model ... 54
4.4.5 Complications ... 60
4.4.6 Implementation of Hierarchical Scheduling ... 63

Chapter 5 Performance Measurement and Analysis ... 65
5.1 Test Case ... 65
5.2 Performance Results ... 66
5.2.1 Performance of the Test Case ... 67
5.2.2 Worst Case Response Time ... 67
5.2.3 Worst Case Interrupt Disable Time ... 68
5.2.4 Communication Times ... 69
5.2.5 Interrupts ... 71

Chapter 6 Conclusions ... 72

6.1 Future Work ... 73

References ... 75

Appendix A Definitions in Apertos ... 79

A.1 Definitions Relating to the Object Paradigm ... 79

A.2 Definitions Relating to Real-time Criteria ... 82

A.3 Definitions Relating to Distributed Systems ... 83

Appendix B Steps in Communications between Active Objects in Apertos ... 84

B.1 Execution Associated with a Requesting Object ... 84

B.1.1 Finding the Target Object ... 85

B.1.2 Asynchronous Send or Synchronous Call ... 86

B.1.3 Return from a Synchronous Call ... 87

B.2 Execution Resulting From a Call to Context::M() ... 88

B.3 Execution within Reflector MMiniMeta ... 89

B.3.1 MMiniMeta::Call() ... 90

B.3.2 MMiniMeta::Reply() ... 90

B.3.3 MMiniMeta::Send() ... 91

B.3.4 MMiniMeta::Find() ... 91

B.3.5 MMiniMeta::reschedule() ... 92

B.4 Execution Within Reflectors MZero and MCore ... 92

B.4.1 MZero::Exit() ... 93

B.4.2 MZero::Find() ... 93

B.4.3 MZero::reschedule() ... 93

B.4.4 MCore::Exit() ... 94

B.5 Execution Resulting From a Call to Context::R() ... 94

Appendix C Adding a New Active Object to the System ... 96

C.1 Core Active Objects ... 96

C.1.1 Object Code ... 96

C.1.2 Object Interface ... 97

C.1.3 Introducing the Object ... 98

C.2 Non-Core Active Objects ... 98

C.2.1 Object Code ... 99

C.2.2 Object Interface ... 99

C.2.3 Introducing the Object ... 100

Appendix D Sample Code ... 102

D.1 A Real-time Clock ... 102

D.1.1 Clock.h ... 102

D.1.2 Clock.cc 103
D.1.3 ClockMsg.h 107
D.1.4 ClockMsg.mc 110
D.1.5 Alarm.h 113
D.1.6 Realtime.h 113
D.2 A User Object 116
D.2.1 ThesisTest.h 116
D.2.2 ThesisTest.mc 116
D.2.3 ULike.h 118

List of Tables

Table 2.1 Comparison of Operating Systems Supporting Objects 23
Table 3.1 MMiniMeta and MCOOP Interfaces 35
Table 3.2 MDrive Interface 36
Table 3.3 MZero Interface 37
Table 3.4 MCore Interface 37
Table 3.5 MetaCore Interface 38
Table 3.6 Exec Interface 38
Table 3.7 Namer Interface 39
Table 3.8 Segment, SystemSegment Interfaces 39
Table 3.9 Mapper Interface 40
Table 3.10 Promise Interface 40
Table 3.11 Idle Interface 40
Table 4.1 Clock Object Interface 45
Table 5.1 Comparison of Communication Times 67
Table 5.2 Costs of Interrupt Handling 71

List of Figures

Figure 1.1 Entities in the Object Paradigm 6

Figure 2.1 COOL v2 Architecture. 14

Figure 2.2 PM System Architecture 18

Figure 3.1 Overview of Apertos Architecture. 25

Figure 3.2 Objects in Apertos 27

Figure 3.3 Reflector Class Hierarchy 29

Figure 4.1 Execution Flow for a Typical Interrupt Handler 42

Figure 4.2 Definition of Hierarchical Scheduling 55

Figure 4.3 2 Dependent Metaspaces Managing 3 Objects 56

Figure 4.4 Proposed Real-time Scheduling Policy in the Meta-hierarchy 59

Figure 5.1 Communication Diagram of Test Case 65

Figure 5.2 Original Control Path for Communication in Apertos. 69

Figure 5.3 Control Path for Communication in Apertos using Hierarchical Scheduling 70

Figure B.1 Graphical Representation of Communications Discussed. 85

Acknowledgments

I would like take this opportunity to thank my supervisor, Dr. G. C. Shoja, for his advice during my entire M. Sc. program at the University of Victoria. Dr. G. C. Shoja suggested this research topic, and I am grateful to him for his continuous guidance, encouragement and seemingly endless patience throughout the development of this thesis.

I would also like to thank Dr. E. G. Manning and K. Murata for their valuable suggestions regarding the experimental work.

In addition, I would also like to thank members of Sony CSL, Tokyo, for providing source materials and hardware regarding the Apertos project, as well as their continuous cooperation.

I would like to thank NSERC for funding for this degree.

Finally, I would like to thank my parents, for without them, this would not have been possible.

Chapter 1

Introduction

This thesis describes mechanisms devised and implemented to enhance the overall, and specifically, real-time performance of a distributed, object-oriented operating system called Apertos. Apertos makes vast use of the object-oriented paradigm throughout its design; the operating system employs system objects to directly support application objects. However, the design incurs substantial performance penalties. To address this performance penalty, the notions of preemptive scheduling and hierarchical scheduling are introduced as solutions.

Before Apertos and the changes introduced to it are discussed, we present an overview of the object-oriented paradigm.

1.1 Object-Oriented Paradigm

Object-oriented approaches have been the focus of much study since the early 1980s. In the object-oriented approach, a system is based on objects, actions, and their relationships. At the conceptual level, an object is an autonomous entity that communicates by message passing with other objects. Object-oriented techniques have shown promise in producing more reliable, easier to maintain code with shorter development times. This is mainly due to their nature, which combines modularity, encapsulation, information hiding, and reusability [1].

Object-orientation is a technique for structuring problems by dividing them into

entities and relations among them, thus reducing the complexity of the problems [2]. In non-object-oriented systems, different programming language elements are named with different terms. There are types, variables, and procedures at the program level, and processes running these programs at the system level. In object-oriented systems, all entities are objects. Analytical descriptions are now made of classes and objects throughout the design and implementation of the system. Even the running system consists of objects, supported by system objects.

A major benefit of the object-oriented paradigm is that it allows the evolution of a system in terms that are more understandable to application and system programmers, as well as to the end user. Another advantage is that the implementation details of an object can be changed without affecting the rest of the system, thereby increasing maintainability. *Inheritance* is a primary difference between the object paradigm and others, such as structured programming, and is a powerful feature that provides for the reusability and extensibility of software components [3]. Finally, because an object in an object-oriented system encapsulates its data and actions, objects are similar to abstract data types.

For clarity, definitions for *class*, *object*, and *metaobject* are given in the following sections.

1.1.1 Class

In the model used in this thesis, the notion and features of a class are described as follows:

- A class can be thought of as a template which is used as a specification for creating new objects. This includes obvious components such as constructors and destructors, but also includes information such as how much memory an object will require.

- A class is defined as a static and immutable entity during run-time, so that it is easy to manage class duplication.
- A class can be defined as a *subclass* of other classes, either by single or multiple inheritance. Not all languages support multiple inheritance. A class can inherit properties from its *superclass(es)*.
- Incremental programming is encouraged using the superclass/subclass relationship. The superclass/subclass relationship defines the *class hierarchy*.
- The code for *methods* defined in a class can be shared among the objects (i.e. instances) of the class. A method defines an action which can be performed by an instantiation of the class. Constructor and destructor methods may be defined, and are invoked when an instantiation of the class is created and destroyed, respectively.

A class is also an *object* (defined in the next section) which is created and managed by a *meta-class*. The meta-class defines the computation for classes. Smalltalk is an example of a system which follows this philosophy. For more information regarding the notion of a meta-class, please refer to [4].

1.1.2 Object

An object is an instantiation of a class. As such, it maintains its own storage for computation, which is to say it is granted singular access to the data which comprises the object (even though languages such as C++ allow this access can be overridden by the class definition). An object also has control over any actions to be performed during initialization, as well as any actions to be performed when the object is destroyed. Any number of objects can be instantiated from a single class.

The computation of an object is defined and managed by its *metaobjects*, which

constitute its *metaspace*. Metaobjects can also create and destroy objects. A metaobject is also an object, which has its own metaobjects. This symmetry creates a *meta-hierarchy* which conceptually can be continued infinitely. Some systems, such as Apertos, limit this hierarchy to three levels, which roughly correspond to the kernel, the operating system, and the application levels of a traditional system. Other systems, like PM [5], impose no limit, though the operating system is defined in a bounded number of levels.

A *passive object* fits closely to the definition of an object. It does not address synchronization or mutual exclusion problems, and has no intrinsic thread of control associated with it.

An *active object*, or *concurrent object* is an object which encapsulates local storage and methods with a virtual processor. Exactly one thread of control is associated with a concurrent object at any time. An active object addresses synchronization problems; simultaneous requests are synchronized at the entry point of the object [6].

1.1.3 Metaobject

Metaobjects have the following properties [4, 7]:

- An object is created and destroyed by its metaobjects.
- Execution of an object's class methods is conceptually defined by its metaobjects. While the class defines the actions to be performed in the method, the metaobjects define how class actions are to be performed.
- The semantics of communication between objects are defined by the metaobjects.
- The metaobjects are responsible for providing an object with local storage having any desired properties, such as stable storage.
- The object/metaobject relationship defines the *meta-hierarchy.*

An object's computation as defined by its class is referred to as *base-level computation*. A metaobject's computation is referred to as *meta-level computation*. Since a metaobject knows the status of its associated objects, the computation of an object can be *reflected* by the metaobject at any time. *Reflection* is the means by which an object communicates with its metaobjects, and is performed via an object's reflector [8]. Conversely, since the status of the metaobject cannot be known by objects it supports at any given time, an object must issue a request to the metaobject to obtain status information. This rule is analogous to an application program querying its underlying operating system for internal system state information, or a debugger knowing the state of a program being monitored.

The class hierarchy is independent of any object meta-hierarchy. Since a class is a static template, it is a programming and compile-time entity. Therefore, a class is independent of the level in the meta-hierarchy where an object is instantiated and placed. This is illustrated in Figure 1.1. The meta dependencies between active objects and their metaspaces, and between classes and the meta-class, show paths of reflection. In the figure, objects in metaspace *A* are metaobjects to the objects supported by metaspace *A*. Objects in metaspace *A* are supported by metaspace *B*. Multiple inheritance is not illustrated in Figure 1.1.

An object communicates with its metaobjects using reflection [8]. The reflective computation of an object is accomplished either implicitly or explicitly. Inter-object communication and paging facilities for the local storage of an object are examples of *implicit reflection*, while an object sending a message to a metaobject to specify real-time constraints is an example of *explicit reflection*. More detailed definitions the object-oriented paradigm are given in Appendix A.

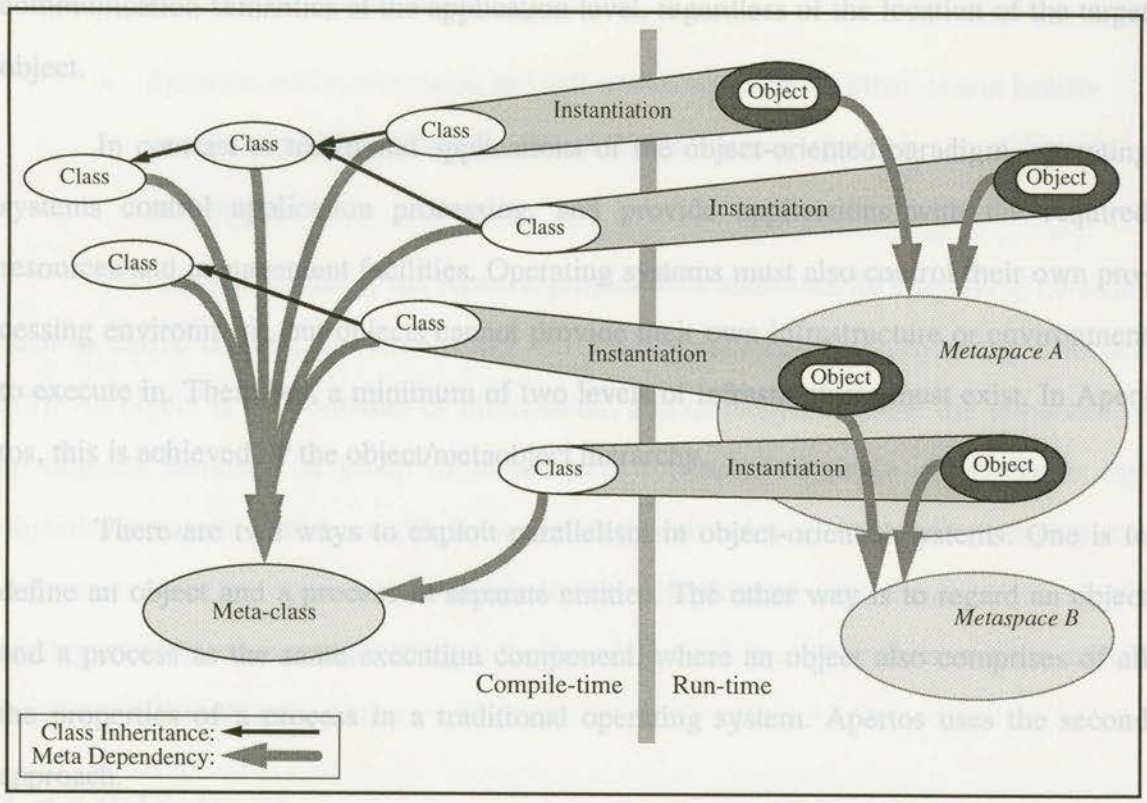


Figure 1.1 Entities in the Object Paradigm

1.2 Object-Oriented Operating Systems

Initially, the use of object-oriented techniques was limited to the implementation step in the development of user applications through the use of object-oriented languages such as Smalltalk and C++. Currently, object-oriented analysis techniques are being used at earlier stages in the object-oriented software development process, including the development of operating systems comprising of, and supporting objects.

Because object-oriented and distributed systems share similar properties, the combination of these is somewhat natural. An object in an object-oriented system has an inherently suitable form for a distributed system [9]. A distributed system consists of multiple autonomous processors that do not share primary memory, but cooperate by messages over communication networks [1]. Object-orientation can provide consistent

communication semantics at the application level, regardless of the location of the target object.

In contrast to traditional applications of the object-oriented paradigm, operating systems control application processing, and provide applications with the required resources and management facilities. Operating systems must also control their own processing environment, but objects cannot provide their own infrastructure or environment to execute in. Therefore, a minimum of two levels of infrastructures must exist. In Apertos, this is achieved by the object/metaobject hierarchy.

There are two ways to exploit parallelism in object-oriented systems. One is to define an object and a process as separate entities. The other way is to regard an object and a process as the same execution component, where an object also comprises of all the properties of a process in a traditional operating system. Apertos uses the second approach.

1.3 Apertos

The object-oriented paradigm and object-oriented systems have been the focus of much research in recent years. Despite their popularity, object-oriented systems have usually been implemented on top of traditional operating systems where little support is provided for objects at the kernel level. The Apertos project [10], undertaken at Sony Computer Science Laboratories (Sony CSL), Tokyo, was among the first attempts at a completely object-oriented operating system. In Apertos (originally called Muse), even system and kernel level support is provided for objects, by objects.

The initial goals of the Apertos project included:

- open-endedness of the system,
- dependability of the execution environment,

- facilities for mobile computing, and
- dynamic self-modification and self-advancement to facilitate object heterogeneity and migration to different metaspaces, possibly to different machines [11].

These initial goals for the Apertos project were addressed by employing concepts such as active objects, and introducing an object/metaobject hierarchy using reflection [11]. An object is the container of information, and metaobjects define the semantics of an object's behavior. A group of metaobjects supporting a single object forms that object's *metaspace*, which roughly corresponds to a virtual machine or an optimal operating environment for objects. All communication is performed using message passing via metaspaces, where messages trigger the invocation of the appropriate method of the receiving active object.

1.4 Objectives

The resulting Apertos system is generic, flexible, and robust. However, these desirable features, while helpful in application development, simulation studies, and research work, impose serious performance penalties such as poor real-time response, heavy communication overheads, and the lack of the ability to withstand a failed process.

The primary objectives of this work are:

- to investigate and implement mechanisms to improve the execution performance of Apertos in general; and
- to identify the causes of the poor real-time performance in Apertos, and provide solutions to improve it.

To achieve these objectives, we have focused our research efforts on the following areas:

- introducing support for timing and real-time services; and
- providing more control over scheduling, with the introduction of preemptive and hierarchical scheduling.

1.5 The Remainder of this Thesis

Chapter 2 is a detailed comparison of Apertos with other operating systems supporting the object-oriented paradigm, as well as those using more traditional approaches to operating system design. An overview of the basic structure and methodology, as well as the primary components of the current implementation of Apertos, are presented in Chapter 3. Chapter 4 discusses major enhancements made to Apertos, with an emphasis on the introduction of preemptive scheduling and hierarchical scheduling. These schedulers contain facilities to support real-time constraints. Chapter 5 presents performance measurements and analyses. Chapter 6 concludes this thesis.

Chapter 2

Background and Related Work

2.1 Overview of Apertos

The Apertos project [10] at Sony CSL, Tokyo was among the first to attempt to build a completely object-oriented operating system, where even system and kernel level support is provided for objects, by objects. Apertos is also the first operating system design based on reflective computing [6]. Several distributed operating systems, such as Clouds [12, 13], Amoeba [14], and Choices[15], have been designed and implemented in the past. These systems have some common features in that they are open-ended, are composed of collections of processes which can be created at a low cost, and that their kernels are minimal. However, these systems have not considered the programmer's point of view in that the systems cannot manage all their resources uniformly, and that they cannot always satisfy the evolving requirements from various kinds of users [4].

2.1.1 Goals of Apertos

Apertos attempts to address these problems by providing a high level of abstraction to users. Some goals when defining the system model are:

- *The system should provide a uniform perspective to users.* Traditional operating systems provide multiple perspectives and definitions, such as files, processes, messages, etc., to users and programmers. The uniform perspective in Apertos is provided by the object definition. All resources are

encapsulated as objects. There are no distinctions between processes and files, or between active and passive objects. All interaction between objects is accomplished by message passing.

- *The system should be self-advancing.* In order to meet the wide variety of requirements demanded by different applications, system parameters and components can change dynamically in accordance with the behavior of executing applications, to improve the performance of the system. Much of this flexibility is realized through the causal connection between an object and its metaobject, which allows an object to change metaspaces (object migration) in order to change and thus improve its execution environment [4, 11, 6]. An object is only causally associated with its metaobjects so that metaobjects can be replaced without having adverse effects on the operation of the object. Reflection is the means by which an object queries and communicates with its metaspace, and from there with other objects in the environment.
- *The system should provide a basis for future operating system development and research.* This capability for expansion exists through Apertos' ability to be self-advancing. New modules can be added and tested without adversely affecting the performance of the rest of the system.
- *The system should be operable in a heterogenous environment, possibly supporting mobile hosts.* Services should be provided in a reliable, secure manner, regardless of the physical location of resources and hosts in the network.

2.1.2 Communication Model

There are different communication models which can be applied to object-oriented systems. For systems to support concurrency between multiple active objects, there

are four strategies [16]:

- *spontaneous activity without receiving a message*. This technique has to be undertaken in a controlled manner. One example of spontaneous activity is the actions that take place upon the creation of a new object. Apertos does provide functionality for this feature, but is seldom used because previous versions of Apertos did not allow communication with other objects from within the constructor. Communication from within the constructor is now supported.
- *allow the invoked object to continue execution after returning results*. This technique requires the initiation of an independent thread of execution for the invoked object which can continue after returning results to the calling object. In Apertos this action is possible, even though it presently isn't used.
- *allow the invoker to proceed with execution in parallel with the invoked object*. Apertos supports this operation. This is analogous to the asynchronous, or nonblocking send in traditional operating systems.
- *invoke methods of several objects at once*. To support this technique in Apertos, many asynchronous messages are sent simultaneously. The basic requirement for this is a mechanism by which the invoker can synchronize with the arrival of a result at any time after making the invocation. Ideally, the invoker blocks only when it requires the return value from a remote method invocation which has not yet completed. Cognac [17, 18] is a language which supports this by definition.

In practice, however, the use of synchronous, or blocking calls, combined with the fact that an active object in Apertos is single threaded and non-preemptable [19], reduces Apertos to a sequential object-oriented model as defined in [16] - where no func-

tionality would be lost if the system were reduced to a single thread of control. With the introduction of preemptive scheduling, however, the asynchronous message send provided by Apertos can be used as it is intended, to provide true concurrency between active objects. The introduction of preemptive scheduling to Apertos is discussed in Chapter 4.

2.2 Other Operating Systems Which Directly Support Objects

To appreciate the reasons behind the decisions taken in designing Apertos, it is helpful to examine other object-oriented operating systems.

2.2.1 Chorus

The *Chorus Object-Oriented Layer* (COOL) [20, 21, 22] is built above the Chorus micro-kernel to extend its micro-kernel abstractions with support for object-oriented systems. Specifically, it provides generic support for *clusters*, which are groups of objects, in a distributed virtual memory model. COOL can be divided into three layers:

- The *COOL-base* is the system level layer. Its interface is a set of system calls which encapsulates the Chorus micro-kernel. It presents itself as a micro-kernel for object-oriented systems, on top of which the *generic run-time layer* can be built. The abstractions implemented in this layer have a close relationship with Chorus itself. To address the poor performance of the initial work, this layer now limits management to *clusters* and *cluster spaces*, rather than objects themselves. A cluster is viewed as a place where related objects exist, similar to an execution environment defined by an Apertos metaspace.
- The *COOL generic run-time layer* is responsible for object management, including creation, dynamic linking and loading, and transparent invoca-

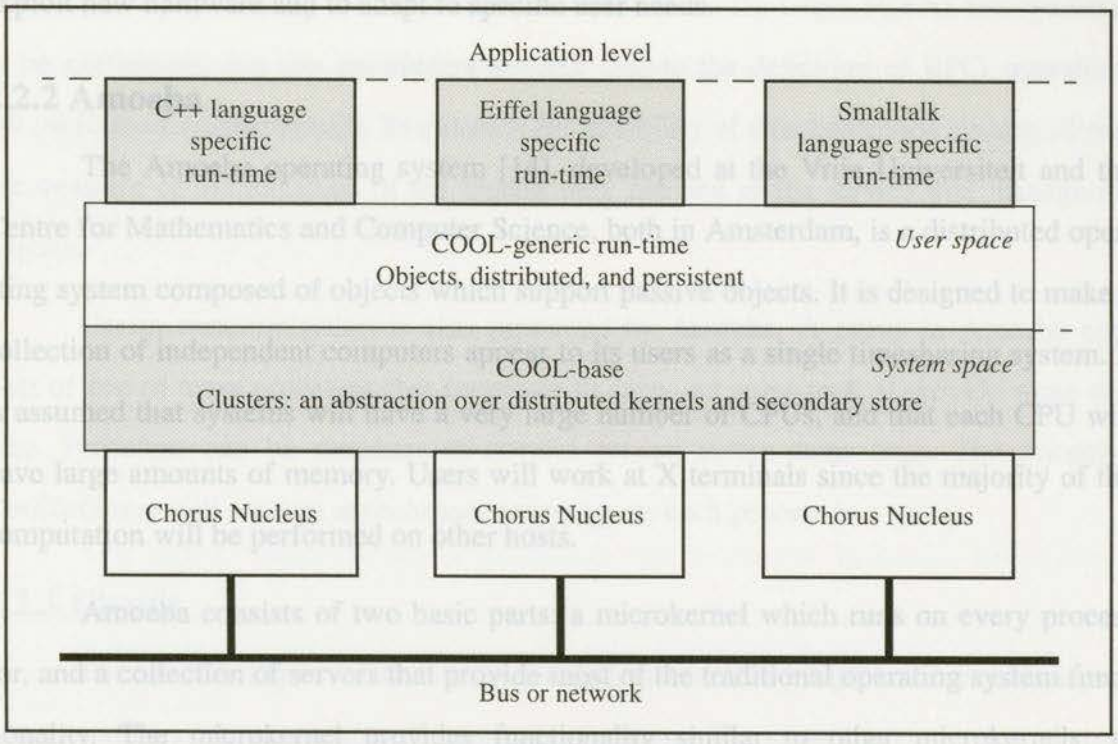


Figure 2.1 COOL v2 Architecture

tion including the retrieval of frozen objects from secondary storage, and the mapping of objects into cluster spaces. Two types of identifiers must be provided at this layer: *domain wide references* which are globally unique, and *language references* which are valid only in the context into which the object is presently mapped.

- The *COOL language-specific run-time layer support* maps a particular language object model to the *generic run-time model*. This activity may be performed by using pre-processors to generate the correct stub code and the use of an upcall table.

The COOL model allows for operating system service objects to be invoked, migrated between address spaces, and moved to persistent store in a manner similar to that used for user objects. It has the ability to dynamically add services to a currently active operating system, and to reconfigure Chorus and/or COOL during run-time to

exploit new hardware and to adapt to specific user needs.

2.2.2 Amoeba

The Amoeba operating system [14], developed at the Vrije Universiteit and the Centre for Mathematics and Computer Science, both in Amsterdam, is a distributed operating system composed of objects which support passive objects. It is designed to make a collection of independent computers appear to its users as a single timesharing system. It is assumed that systems will have a very large number of CPUs, and that each CPU will have large amounts of memory. Users will work at X terminals since the majority of the computation will be performed on other hosts.

Amoeba consists of two basic parts: a microkernel which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality. The microkernel provides functionality similar to other microkernels: to manage processes and threads, provide low-level memory management, support communication, and handle low-level I/O.

Server processes perform everything that is not done by the kernel. This division is to enhance the flexibility of the overall system, so that multiple versions of a service can be run simultaneously, and so they can be changed easily. The basic unifying concept underlying all the Amoeba servers and the services that they provide, is the object.

When a process creates an object, the server that manages the object returns a *capability* for the object to the client. For the client to use the object later, a valid capability must be presented. Capabilities are used for both object naming and protection. This abstraction provides run-time support analogous to, but much more powerful than, the C++ compile-time modifiers *private*, *protected*, and *public*, which are used within a class definition.

To perform an operation on an object, a client invokes a *Remote Procedure Call*

(RPC) [23] by specifying the capability (which identifies the target object), the operation to be performed, and any parameters needed. Due to the definition of RPC, operations are performed synchronously. To enhance the flexibility of this distributed system, clients are unaware of the locations of the objects they use and of the servers that manage the objects.

Group communication is also supported by Amoeba. A *group* in Amoeba consists of one or more processes that cooperate to carry out some task or provide some service. Processes can be members of several groups at the same time. The primitive *SendToGroup* will send an asynchronous message to each process in a group.

2.2.3 Clouds

Clouds [12, 13] is a distributed operating system that integrates a set of independent computers into a conceptually centralized system. The system is composed of compute servers, data servers, and user workstations. A *compute server* is a machine that is available for use as a computational engine. A *data server* is a machine that functions as a repository for persistent data. A *user workstation* is a machine that provides a programming environment for developing applications, and an interface to the compute and data servers executing those applications. The structure of Clouds promotes location, access, concurrency, and failure transparency.

Clouds is implemented upon a kernel called *Ra*, which is a native kernel running on bare hardware. Clouds is designed using an object/thread model. All instances of services, devices, programs, and data are encapsulated as passive entities called objects. Threads are the only active entities in the system, and are used to execute the code in an object.

Local and remote (RPC-based) invocations of objects are differentiated by Clouds, but are transparent at the application level. Distributed shared memory allows remote objects to be moved to a local node so that an invoked object may execute locally.

Clouds provides system researchers with an adaptable, high-performance workbench for experimentation in distributed computing. One disadvantage of Clouds, however, is that there is no way to subclass Clouds objects. Inheritance is not supported.

2.2.4 PM Operating System

The PM distributed object-oriented operating system is a project at the Friedrich-Alexander University, Germany [5, 24, 25] with many similarities to Apertos. In the abstract sense, the PM project consists of two major parts: the *object model* and the *system architecture*.

The object model defines a distributed object-oriented language that contains abstractions for distribution, concurrency, and security. The system architecture then uses the object model to implement the components of the operating and run-time systems.

Both the objects that implement the operating system services and those that use its features use the same object model, so the interfaces are compatible. Since the operating system is not one monolithic block, as are many traditional operating systems, such as UNIX, any application can build operating system components on top of existing abstractions for services and hardware. In PM, these abstractions are called *meta-layers* which require interfaces between an application system and the meta-system. This relationship directly corresponds to the object/metaobject relationship using reflectors in Apertos.

Virtual address spaces are called *object spaces*. Objects can be mapped into several object spaces simultaneously. Operating system objects can be placed in the application object space, and application objects can be mapped into an operating system object space to be able to use e.g. CPU supervisor mode. Method calls across object space borders are more expensive than local method calls. An object space in PM roughly matches the metaspace provided by Apertos, but while Apertos emphasizes the ability of objects to migrate between metaspaces, PM doesn't. When an object is placed on one meta-layer in PM, it remains there for the duration of its existence.

2.2.5 Spring

Spring [26, 27] is a distributed operating system that is focused on developing well defined interfaces between OS components. The combination of well defined interfaces and its object-oriented approach helps to achieve goals such as openness, extensibility, easy distributed computing, and security.

The operating system is based upon a minimal kernel. Only two components of the operating system run in kernel mode: the *nucleus,* which manages processes and interprocess communication, and the *virtual memory manager,* which controls the core facilities for paged virtual memory. All other system services, such as naming, networking, and file systems, are implemented as user mode services on top of the kernel. These services are built as dynamically loadable modules; the operating system decides which are loaded at boot time.

The Spring multithreaded microkernel supports three basic abstractions:

- A *door* is a communication endpoint, to which threads may execute cross-address-space calls. Parameters and return values are passed through doors.

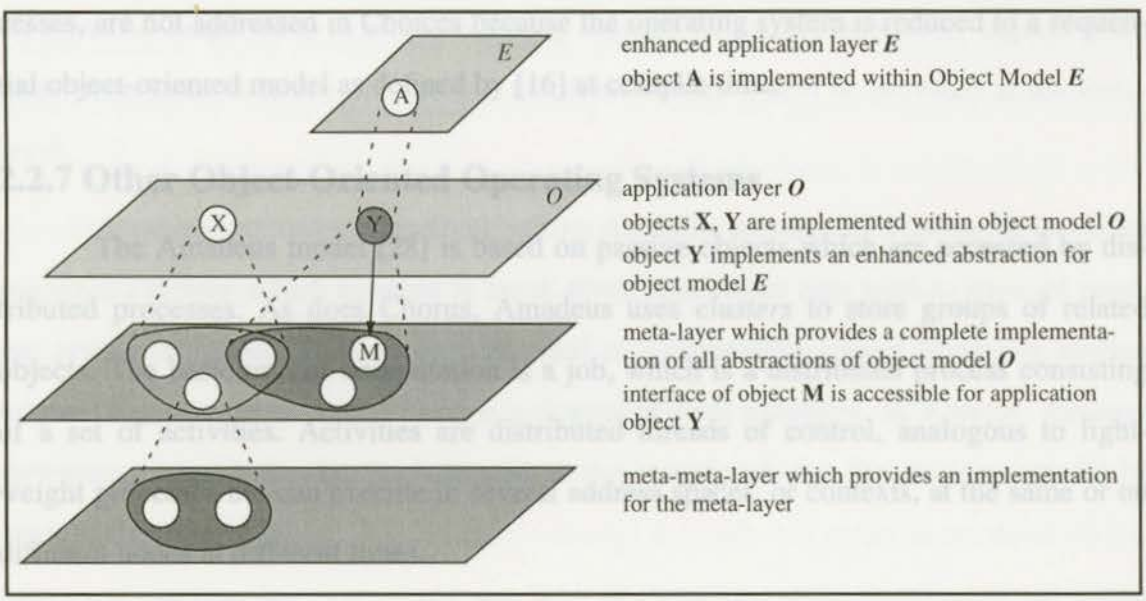


Figure 2.2 PM System Architecture

- A *domain* is a logical grouping of related objects, often sharing the same address space. A domain can have multiple threads.
- *Threads* are represented as nucleus objects, accessible through cross-domain calls into the nucleus.

2.2.6 Choices

The Choices operating system [15, 2] is implemented as a framework of C++ classes, and was developed at the University of Illinois. This operating system uses a language approach, where object-orientation is only applied to the production of code by using the features of the language. Consequently, there may not be any object-oriented structure at run-time.

Applications use system services by creating instances of system classes, such as files, and accessing them by their methods. A proxy mechanism is used to cross the user/system barrier. This approach makes use of the object-oriented facilities of C++ at the programming level. The executing kernel is monolithic, and objects are simply "portions of data in memory", and therefore are passive, as defined in C++. Activities, or processes, are not addressed in Choices because the operating system is reduced to a sequential object-oriented model as defined by [16] at compile time.

2.2.7 Other Object-Oriented Operating Systems

The Amadeus model [28] is based on passive objects which are accessed by distributed processes. As does Chorus, Amadeus uses *clusters* to store groups of related objects. The basic unit of computation is a job, which is a distributed process consisting of a set of activities. Activities are distributed threads of control, analogous to lightweight processes but can execute in several address spaces, or contexts, at the same or on different nodes at different times.

Raven [29] is both a distributed operating system and a programming language. It

utilizes a simple, uniform passive object model in which all entities, at least conceptually, are objects, similar to that in Objective C or Smalltalk. Both static and dynamic typing, as well as static and dynamic method binding, are supported in this compiled language. Support for concurrency and distribution are also part of the language. In Raven, it is possible to create multiple threads of control. Each thread can access any object. If more than one thread accesses the same object, concurrency control is used to coordinate access.

ES-Kit is a set of software and hardware building blocks that may be readily assembled to produce a heterogeneous, object-oriented distributed system [16]. The operating system consists of a kernel and a set of *Public Service Objects*. ES-Kit provides an interesting feature not seen in many other object-oriented systems: *method locking*. Method locking primitives allow an object to selectively disable or enable the execution of its methods based on its internal state. This is a powerful tool because it allows an object to overrule, based on its dynamic state, the guarantee provided by the underlying system that method execution is strictly in order of message arrival at the object. For example, if one considers a device driver object handling both input and output, requests for input no longer need to block following requests for output from other objects when there is no input to return. In Apertos, this functionality would be provided at the meta-level.

A natural extension of method locking would be *message forwarding*. If an object manages a printer, which happens to break down, that object may wish to have all pending print jobs (message requests) forwarded to a compatible object managing another printer. This operation would provide a degree of fault tolerance. In Apertos, message forwarding would also be implemented at the meta-level, but the decision would be made at the base level.

BirliX [30, 2] supports distributed applications that are structured as collections

of *abstract data type instances* (ADT-instances). Each ADT-instance is an object which runs in a *team* environment. A team consists of activities (light-weight processes, or threads) and resources, and is equivalent to a multi-threaded process with its own address space. ADT-instances communicate via RPC. Experiments on implementing a scheduler in user space in BirliX have also been performed. But because changing between address spaces is only possible in the kernel, the context switching portion of the scheduler remains in the kernel. The decision making portion of a given user scheduler is retained in user space, in memory shared by all schedulers. A master scheduler negotiates with requesting processes and is responsible for creating and managing a new, dependent scheduler to manage those processes.

Oisin [31] is a distributed operating system kernel that supports the object-oriented style of programming, and provides transparent access to both local and remote passive objects. Oisin is similar to Chorus and Amadeus in that it employs clusters, contexts, and jobs. Although contexts may not be shared between jobs, objects may be shared between contexts of different jobs at the same node.

Angel [32] moves away from message passing paradigm in favor of a SASA, or *Single Address Space Architecture*. Through testing, it was found that the typical series of steps used for traditional message passing required too much overhead. To address this overhead, Angel utilizes shared memory where all processes, or objects, share a single address space. The memory protection mechanism is not part of the process, but is performed at the address translation stage. Objects are used as a tool to allow/disallow other objects access to virtual memory pages. Using this technique, performance has increased significantly while maintaining all the functionality provided by a traditional kernel, including protection between objects. Note that this architecture design is not intended to support shared variables (even though it could), but rather to cut down on the traditional message passing overheads, such as copying and/or re-mapping parameter

and return values between the address spaces of the requesting and target objects, and any system levels involved.

Because of the high communications overhead seen in Apertos, a single address space for all objects or groups of objects should at some time be considered in hopes of achieving similar performance gains. It may be reasonable for all objects sharing one metaspace to share the same address space to further optimize communication.

The Mach kernel of CMU [33, 2, 34] only provides the infrastructure for the existence of and communication between server processes. It is these server processes which actually provide the operating system functionality. Outside the kernel, these server processes, which are instances of services, are called objects. Some commercial systems are based on such sets of instances, such as OSF/1 and NextStep [2]. The UNIX interface can be fully provided by sets of servers.

2.3 Comparison

Operating System	Type of Object Supported	Support for Persistent Objects	Support for Real-time
Amadeus	passive objects	provided as files, database entries	unclear, likely not
Amoeba	passive objects	provided as files	none
Angel	passive objects	provided by SASA memory sub-system	not stated
Apertos	active, passive objects	provided by meta-objects	provided by meta-objects (being added)
BirliX	active teams each comprising of a passive object and a process	provided by system	support for schedulers in user space, real-time
Choices	passive objects	provided, persistent objects used for communication	supported
Chorus	passive objects	provided by a sub-system	micro-kernel has a real-time scheduling facility
Clouds	passive object, class inheritance not supported	provided by kernel (user transparent)	none
ES-Kit	active, passive objects	provided by public service objects	none
Mach	may support active objects	may be provided by a sub-system	version of kernel available supporting real-time
Oisin	passive objects	not stated	none
PM	active objects	provided by a meta-layer	hierarchical scheduling not supporting real-time
Raven	passive objects	provided by system	none
Spring	passive objects	not stated	none

Table 2.1 Comparison of Operating Systems Supporting Objects

Chapter 3

Current Implementation of Apertos

Apertos is an ongoing research project where the design still changes significantly between versions. For this reason, this chapter only discusses version 0.5.1 of the operating system, the latest publicly available release. Where necessary, changes between previous and present versions have been noted.

Apertos utilizes the object/metaobject hierarchy throughout the entire system. All objects communicate with their metaobjects via their reflectors. Objects can share reflectors, and hence also share metaobjects. Metaobjects themselves are objects, which also require a metaspace to execute upon. Figure 3.1 illustrates a possible configuration of the Apertos system. The arrows in the figure represent paths for reflection, defining the meta-hierarchy.

3.1 Active Objects

One important concept in Apertos is that of *active objects*. An active object is similar to a traditional passive object provided by C++ in that it has a set of methods which can be invoked by other objects, and has private data and methods for exclusive use by the object. An active object, however, has in addition associated with it all the properties of a process in a traditional operating system: a context, communication paths, message queues, etc. Each active object has associated with it a single *Activity* object, which retains all state information.

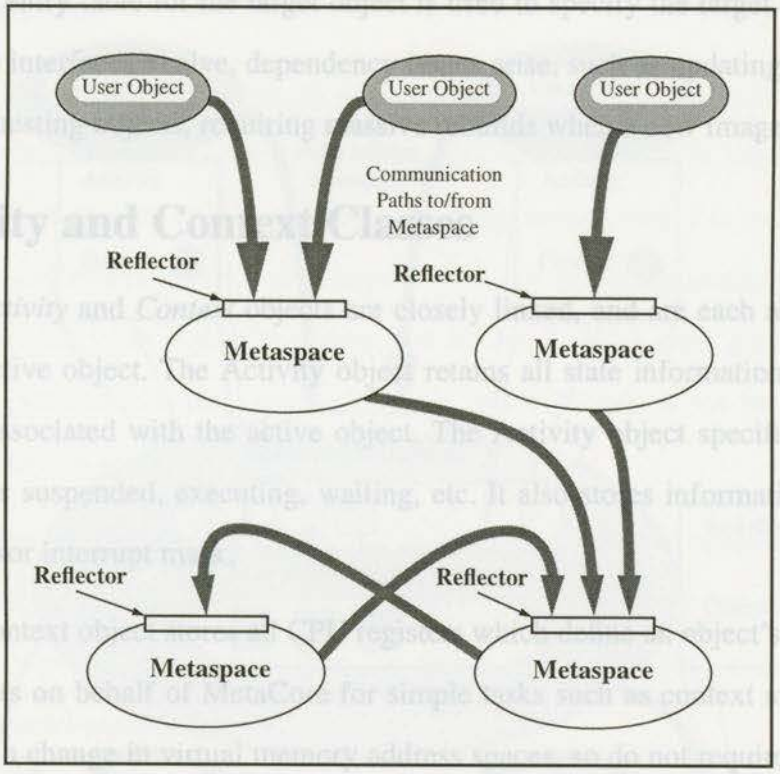


Figure 3.1 Overview of Apertos Architecture

Since an object has system state information associated with it, it is not called directly by a communicating object. Rather, a calling object requests the system to execute a method via a message call directed at the object. This technique is used to provide security and modularity between objects. Publicly accessible methods have stub routines which act as a transition between the system call and the actual method invocation. Stubs unpack messages containing parameters before sending them on to the target method. C++ is used to write both passive and active objects. MC++ [35], however, has the advantage that it can automatically generate the required stub code and message structures for active application objects for Apertos.

Entry points to the stub code are placed in a table, similar to a function table for a dynamically linked library, or a virtual function table for a C++ object. This table is generated for use by an object's meta-level. Methods are numbered, and when an object requests to execute a method associated with another active object, the index of the

method in the entry table for the target object is used to specify the target method. However, as object interfaces evolve, dependency issues arise, such as updating method index values for requesting objects, requiring massive rebuilds when a new image is created.

3.2 Activity and Context Classes

The *Activity* and *Context* objects are closely linked, and are each associated with exactly one active object. The Activity object retains all state information and manages any Context associated with the active object. The Activity object specifies whether its active object is suspended, executing, waiting, etc. It also stores information such as its default processor interrupt mask.

The Context object stores all CPU registers which define an object's run-time context. It also acts on behalf of MetaCore for simple tasks such as context switches which do not require a change in virtual memory address spaces, so do not require a trap to kernel mode (only in the version of Apertos which executes on the MIPS[1] R3000 series processor). The Context structure is owned by MetaCore, even though an object will implicitly access its own Context every time that object wishes to communicate with another active object, including its reflector. For actions which require MetaCore, Context acts as an interface between the object and MetaCore.

With the introduction of preemptive scheduling (discussed in detail in Section 4.2), it became necessary for an object to identify itself when communicating with its reflector. An object's Context structure is used for identification of the object, and resolution of its Activity.

Because the Activity and Context objects contain information such as processor register values, interrupt masks, etc., their definitions and implementations are architec-

1. MIPS is a registered trademark of MIPS Computer Systems

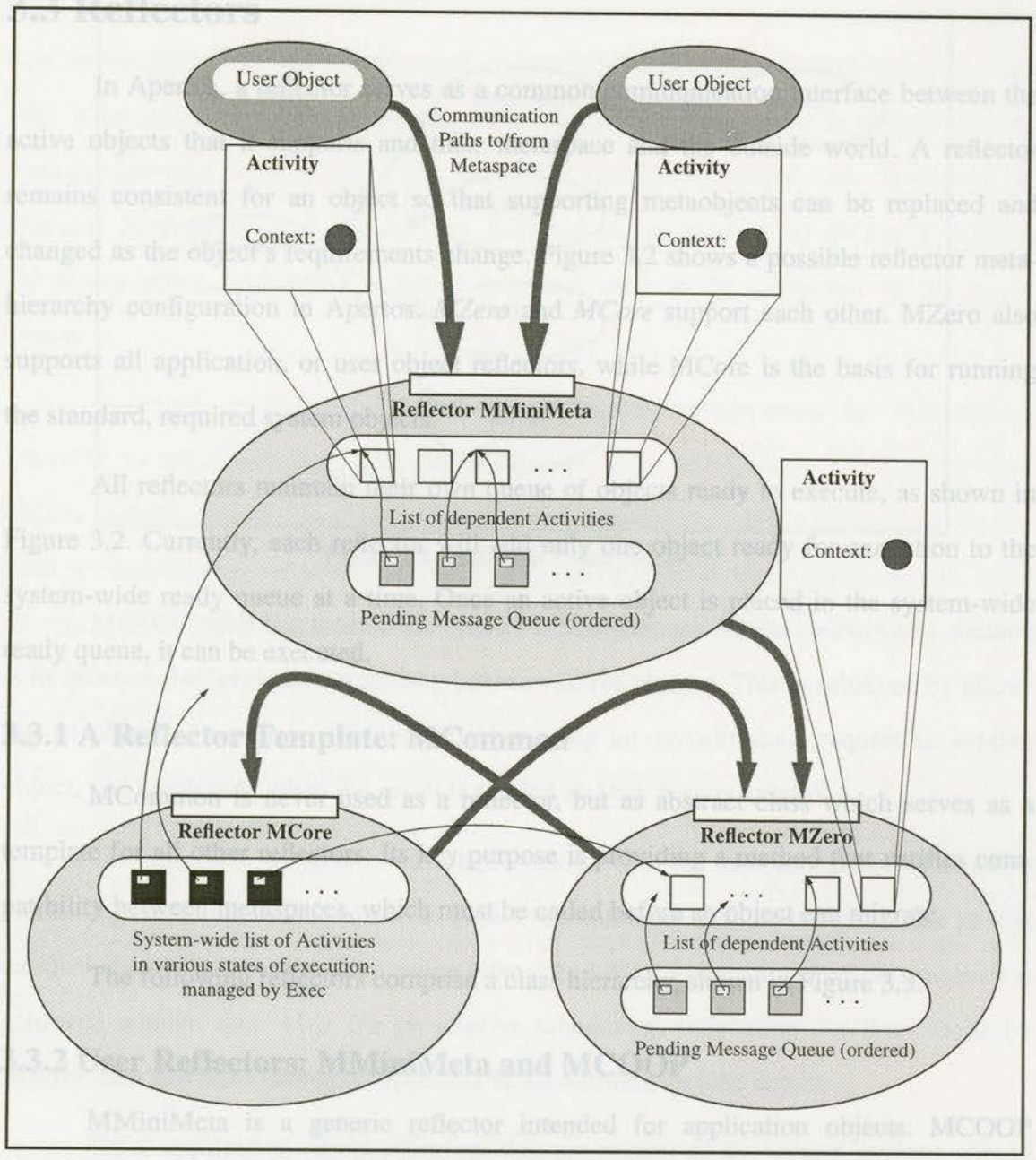


Figure 3.2 Objects in Apertos

ture dependent. Currently, there are implementations of Apertos which execute on the MIPS R3000 processor and the Intel 80486 processor.

3.3 Reflectors

In Apertos, a reflector serves as a common communication interface between the active objects that it supports and their metaspace and the outside world. A reflector remains consistent for an object so that supporting metaobjects can be replaced and changed as the object's requirements change. Figure 3.2 shows a possible reflector meta-hierarchy configuration in Apertos. *MZero* and *MCore* support each other. MZero also supports all application, or user object reflectors, while MCore is the basis for running the standard, required system objects.

All reflectors maintain their own queue of objects ready to execute, as shown in Figure 3.2. Currently, each reflector will add only one object ready for execution to the system-wide ready queue at a time. Once an active object is placed in the system-wide ready queue, it can be executed.

3.3.1 A Reflector Template: MCommon

MCommon is never used as a reflector, but as abstract class which serves as a template for all other reflectors. Its key purpose is providing a method that verifies compatibility between metaspaces, which must be called before an object can migrate.

The following reflectors comprise a class hierarchy, shown in Figure 3.3.

3.3.2 User Reflectors: MMiniMeta and MCOOP

MMiniMeta is a generic reflector intended for application objects. MCOOP (*M*eta for *C*oncurrent *O*bject-*O*riented *P*rogramming) is a subclass of MMiniMeta. These reflectors share essentially the same interface and are compatible with each other, as outlined in Table 3.1. MCOOP's significant addition is support for *continuations*, which allow an active object to call the object which invoked it. To implement continuations, MCOOP redefines the specific actions of `Call()`, `Send()`, `Reply()`, `Start()`, and `Stop()`.

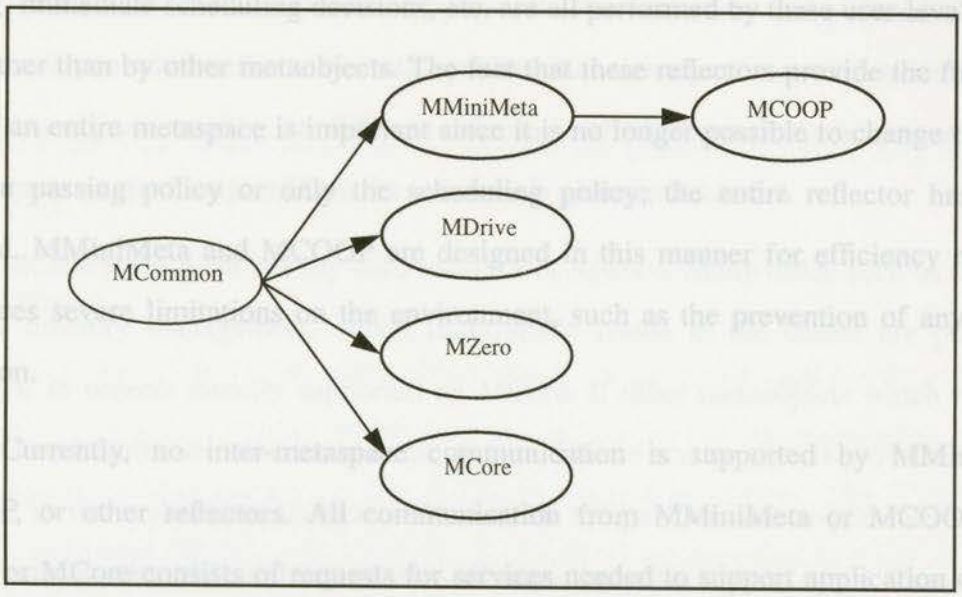


Figure 3.3 Reflector Class Hierarchy

MCOOP adds the method `Receive()`. The purpose for the `Receive()` method is to increase the level of concurrency between active objects. This is achieved by allowing an object to continue executing after sending an asynchronous request to another object, and block only when the result is needed and has not yet arrived.

The `Yield()` method is no longer needed with the introduction of preemptive scheduling. Its purpose is to allow an object to voluntarily give up the processor so that another object can execute. With preemptive scheduling, however, object concurrency is achieved without this. Also, for preemptive scheduling, scheduling decisions made by MMiniMeta had to be changed. Appendix B.3.5 outlines specific details.

Other application object reflectors that are needed as the system evolves should be derived from one of these two reflectors. These define the basic, minimum functionality required by Apertos at this time.

It should also be noted that MMiniMeta and MCOOP presently provide a complete metaspace as needed by a dependent object. Base scheduling and memory management are still performed strictly by metaobjects on MCore, but activities such as message

passing, immediate scheduling decisions, etc. are all performed by these user-level reflectors, rather than by other metaobjects. The fact that these reflectors provide the functionality of an entire metaspace is important since it is no longer possible to change only the message passing policy or only the scheduling policy; the entire reflector has to be changed. MMiniMeta and MCOOP are designed in this manner for efficiency reasons, but places severe limitations on the environment, such as the prevention of any object migration.

Currently, no inter-metaspace communication is supported by MMiniMeta, MCOOP, or other reflectors. All communication from MMiniMeta or MCOOP with MZero or MCore consists of requests for services needed to support application objects. This severely constrains the utility of this version of Apertos for experimentation, since all application objects must then be supported by the same application reflector.

3.3.3 Device Driver Reflector: MDrive

MDrive is a specialized reflector intended specifically for device driver objects, and supports such services as continuations [36]. It has a more comprehensive interface than MMiniMeta or MCOOP, as shown in Table 3.2. Although it does not presently exist for the R3000 version of Apertos, it will be implemented in future releases, as it has been for the i486 version.

3.3.4 System Reflectors: MCore and MZero

For complete consistency in the system, the base of the meta-hierarchy is defined by two reflectors, MCore and MZero. These reflectors provide a space for each other to execute upon, as shown in Figure 3.2. As defined in the object model presented in Chapter 1, each and every object must have a metaspace to execute upon. Therefore, there can not be a single 'base' system object, but rather, the base of the meta-hierarchy must be circular. There exists only one MCore and MZero per system.

MZero is the reflector which supports all other reflectors in Apertos. Like other reflectors, it provides methods for both synchronous and asynchronous communication. Two active objects residing on different reflectors, which share the same parent MZero must communicate through MZero. Its interface is given in Table 3.3.

MCore is the most heavily used reflector in Apertos. Many tasks, such as scheduling and memory management, which traditionally reside in the kernel are placed in MCore or in objects directly supported by MCore. If other metaobjects which redefine any of these features are introduced to the system, MCore and its objects provide a formal interface to the underlying hardware. Because of the performance requirements of the objects managed by MCore, these objects all share the same address space. Communication between objects residing directly on MCore is not limited to MCore's message primitives. For performance, these active objects can communicate as do passive objects. MCore's interface is given in Table 3.4.

3.4 MCore Metaobjects

There is a select number of objects which run on MCore and provide services to the entire system. Only objects which exist both in the i486 and R3000 versions of Apertos are outlined here.

3.4.1 Segment, SystemSegment, and Mapper

Segment is a virtual class which provides a common interface to all memory segments in the system. *SystemSegment* inherits from Segment and is used to represent all the memory segments used by MCore and MetaCore.

Mapper manages virtual paged memory and all its associated structures, tables, objects, and hardware. It currently supports virtual-to-real memory address translation, but does not completely support swapped virtual memory. Each page of virtual memory is mapped to a passive object, called a *Page*, which is owned strictly by Mapper. Each

Page object knows which Segment actually holds the memory. Mapper also provides interfaces that can add or remove associated virtual memory pages for a specific segment.

Interfaces for SystemSegment and Mapper are given in Table 3.8 and Table 3.9, respectively.

3.4.2 Namer

The *Namer* object is an active object which maintains the local (to the computer) list of *S*hort *ID*entifiers (SIDs), which function as handles for communication between active objects. It also manages a list of *O*bject *ID*entifiers (OIDs), *O*bject *AD*dresses (OADs), and *L*ocal *AD*dresses (LADs). These identifiers are discussed in detail in [37].

Essentially, the OID represents the original, and the OAD represents the current, naming context of an object. OIDs are variable length identifiers which roughly follow the hierarchical structure within Apertos, but can be used in a distributed environment. OIDs are relative in the meta-hierarchy from the local Namer. An OAD is a hint to the current location of the object, but doesn't provide the optimal communication path. LADs and SIDs are valid only on the local node. They are used for faster name lookup of objects, and for communication within the system. Active objects need only be concerned with SIDs, while the underlying system must be concerned with OADs and OIDs when communication outside the local computer is required.

Namer's interface is presented in Table 3.7.

3.4.3 Exec

Exec is an active object which executes on MCore. It provides the basic system scheduling policy, which is presently a prioritized FIFO algorithm. With a slight modification, Exec also provides prioritized round-robin scheduling, used for preemptive scheduling.

Exec also retains ownership of all Activity objects in the system, granting access

to them to other objects in the system. As with other objects supported by MCore, methods can be called by MCore or by other objects directly supported by MCore (excluding MZero) as if Exec were a passive object. This access is provided to improve performance and to minimize context switches where heavy communication is involved. Other objects, such as MMiniMeta and MCOOP, must use standard communication primitives to access Exec. Exec's methods are described in Table 3.6.

3.4.4 Promise

Promise is supported by MCore, but is accessible throughout the system. Promise is used in conjunction with continuations to allow objects which use continuations to complete as soon as possible. Its purpose is to provide a facility to add the requesting object to Exec's run queue at the highest priority, to force the system to allow the object to run at the very next possible opportunity. It seems that this functionality should be provided by Exec rather than by this separate active object. Its interface is given in Table 3.10.

3.4.5 Idle

Idle is an object which executes on MCore, and cannot be seen by the rest of the system. Its purpose in the system is simply to ensure that there is always at least one object in Exec's run queue, to simplify the logic for executing objects. It executes at the lowest priority, and thus only runs when no other task is available to run in the system. It exits immediately to allow any other ready object to execute. For completeness, its interface is given in Table 3.11.

3.5 MetaCore

Apertos' MetaCore is analogous to a microkernel. It provides the absolute minimum set of required services, often transferring control to another object which can perform the required tasks. Its interface is very small, taking the model of a microkernel to

its extreme. Table 3.5 describes MetaCore's interface. MCore provides a metaspace for MetaCore.

In practice, an active object's Context structure will act as an interface for the object to MetaCore. This is to minimize CPU traps which result in context switches, and to simplify issues regarding virtual memory spaces. For example, previous versions of Apertos would always use a trap to perform a context switch. In version 0.5.1 of Apertos, traps are only used when required; when there is a change in memory address space.

3.6 Communication between Active Objects

Communication between all active objects, with the exception of some on MCore, is done strictly via their respective meta-layers. Given two application objects sharing the same reflector, the caller sends the message request to the reflector, and the reflector then performs the tasks required to deliver the message, and finally invokes the appropriate method of the callee.

If objects do not share the same metaspace, then messages are passed to some shared meta-layer. For example, when MMiniMeta wishes to add a ready object to Exec, communication must ultimately pass through MCore, a reflector which directly supports Exec and indirectly supports MMiniMeta. Refer to Figure 3.2 to see how MCore supports these objects.

A detailed analysis and walk-through of two communicating objects is presented in Appendix B: Steps in Communications between Active Objects in Apertos.

3.7 Object Interfaces

Method	Description
`Call`	handles a synchronous message send request
`Send`	handles an asynchronous message send request
`Reply`	handles an object's return from a method invocation
`Yield`	suspends current execution to start another, adds suspended object to run queue
`Find`	forwards the request to MZero, which returns an active object's handle (SID), which can be used for communication
`Restore`	initializes and creates the method table which contains entry points into the object; currently only used during system start-up
`RestoreSelf`	initializes the method table for the reflector; currently only done at system start-up
`Start`	interface to a metaobject, currently used only at system start-up
`Stop`	interface to a metaobject
`Receive`	MCOOP only; for receipt of a reply from an earlier asynchronous message send request

Table 3.1 MMiniMeta and MCOOP Interfaces

Method	Description
`Call`	handles a synchronous message send request
`Send`	handles an asynchronous message send request
`Reply`	handles an object's return from a method invocation
`Bind`	interface to MetaCore's Bind facility
`Unbind`	interface to MetaCore's Unbind facility
`Grow`	forwards the request to the underlying system for a new page of memory, or to extend an existing page of memory
`Shrink`	forwards the request to the underlying system to either shrink or return an existing page of memory to the system free pool
`Exit`	handles an object's return from a method invocation, where the caller isn't expecting a return value, such as when an object is invoked from the event of an interrupt
`NewContinuation`	allocates a new continuation to the requesting object, including its associated descriptor
`DeleteContinuation`	deletes a continuation associated with a concurrent object
`Restore`	initializes and creates the method table which contains entry points into the object; currently only used during system start-up
`RestoreSelf`	initializes the method table for the reflector; currently only done at system start-up
`Start`	interface to a metaobject, currently only used at system start-up
`Stop`	interface to a metaobject
`Deliver`	handles the event where its supporting reflector (MZero) requests that a message be delivered to an object that MDrive directly supports
`Reschedule`	forces MDrive to reschedule, which currently adds the next available object which can execute to Exec's run queue
`SetMask`	sets the interrupt mask, et. al. for the requesting object; can be used to specify that an object cannot be suspended during execution by setting the mask appropriately
`Find`	forwards the request to MZero, which returns an active object's handle for communication

Table 3.2 MDrive Interface

Method	Description
`Start`	interface to a metaobject, currently only used at system start-up, by MiniApertos
`Install`	adds an object to MZero's object table, for use by Find()
`Remove`	removes an object from MZero's object table
`Find`	returns a active object's handle (SID), which can be used for communication
`Call`	handles a synchronous message send request
`Send`	handles an asynchronous message send request
`Reply`	handles an object's return from a method invocation
`NewContinuation`	allocates a new continuation for the requesting object
`DeleteContinuation`	deletes a continuation associated with the requesting object
`Yield`	suspends current execution to start another, adds suspended object to run queue
`Exit`	handles an object's return from a method invocation, where the caller isn't expecting a return value
`Restore`	initializes and creates the method table which contains entry points into the object; currently only used during system start-up
`GetDescriptor`	returns a reference to the descriptor for the object being asked for

Table 3.3 MZero Interface

Method	Description
`Call`	handles a synchronous message send request
`Send`	handles an asynchronous message send request
`Reply`	handles an object's return from a method invocation
`Exit`	handles an object's return from a method invocation, where the caller isn't expecting a return value
`Grow`	returns a new page of memory, or extends the length of an existing page of memory
`Shrink`	returns memory to the free memory pool
`Deliver`	handles the event where its supporting reflector (MZero) requests a message to be delivered to an object that MCore directly supports

Table 3.4 MCore Interface

Method	Description
M	suspends current context to send a message to its reflector
R	passes control from a reflector to an active object
CActive	returns the current context
CBind	binds a context and method of an object to an interrupt
CUnbind	reverses the actions of CBind

Table 3.5 MetaCore Interface

Method	Description
New	create a new Activity, complete with a valid Context
Delete	destroy an existing Activity and Context
Run	adds the specified Activity to the prioritized run queue
Stop	removes an Activity from the run queue
Top	returns the next Activity to execute for a specified priority, object is not removed from run queue as it is in Next()
Next	returns the next Activity to execute, regardless of priority, and removes it from the run queue, stores a reference to it in a local variable
CurrentActivity	returns the Activity of the currently executing active object
Reschedule	implements the round-robin extension for preemptive scheduling
ChangeAttribute	changes an Activity's attribute in a system-friendly manner

Table 3.6 Exec Interface

Method	Description
NewID	returns a new SID, unattached to any object; the SID has associated with it a newly allocated OAD and OID
NewIDAddr	updates information for a given SID
DeleteID	removes a SID from the list
AddID	for the given OID, allocates and returns a new SID
SubID	used for object migration when an object is migrated to the local machine
ResetOAD	for a SID, makes the corresponding OAD invalid; would be used to notify to the system of an object migration
SetAOD	for a given SID, updates its OAD to that provided
OIDOf	returns the OID for a SID
OADOf	returns the OAD for a SID
SIDOf	returns the SID for an OID, it may require allocating a new SID if this the first reference to a remote object
SIDOfLAD	returns the SID for an LAD
OADOfOID	returns an OAD for an OID
InitSID	initializes a SID to the provided OID and OAD values

Table 3.7 Namer Interface

Method	Description
Initialize	performs basic initialization for a new memory region, requested somewhere in the system
FaultHandler	called when the virtual memory address causes a fault; currently simply resumes the object which caused the fault
Grow	used to create or expand a system memory region
Shrink	used to destroy or shrink a system memory region

Table 3.8 Segment, SystemSegment Interfaces

Method	Description
`Activate`	validates a virtual memory address, given certain attributes both on structures managed by Mapper and at the hardware level
`Deactivate`	removes a page from virtual memory address table, including all work concerning hardware registers
`MakeInvalid`	makes a virtual address entry invalid at the hardware level
`Install`	installs the virtual memory pages managed by the given segment into the Mapper's hash table
`Remove`	removes a segment and all virtual memory pages which were managed by that segment
`NewPage`	allocates and installs a new virtual memory page; a new page presently attaches itself to a SystemSegment
`DeletePage`	currently does nothing, but should communicate with the associated real memory segment
`FaultHandler`	if the address which generated the fault is valid, a message indicating the fault is sent to the segment owning the page

Table 3.9 Mapper Interface

Method	Description
`Run`	adds the requesting object to Exec's run queue at the highest priority, to force it to be executed as soon as possible
`Register`	registers an object's Activity with itself so that the object can call the Run() method when needed.

Table 3.10 Promise Interface

Method	Description
`Run`	gives up the processor by returning to MCore, forcing it to re-add Idle to Exec's run queue and reschedule

Table 3.11 Idle Interface

Chapter 4

Scheduling

This chapter first reviews the current implementation of scheduling in Apertos, followed by an outline of the changes made to introduce basic round-robin preemptive scheduling, in a manner compatible with the current implementation, and without incurring significant performance overhead. Finally, hierarchical scheduling is introduced as a means to support, on the same machine, the operations of objects with dissimilar scheduling requirements.

4.1 Present Scheduling Policies

Currently, Apertos does not support preemptive scheduling between active objects. To date, the operating system strictly follows a cooperative multitasking model, in the sense that objects must voluntarily give up the CPU in order for the system to support concurrency. In fact, until recently, even exception and interrupt handlers followed this model.

Contrary to traditional operating system and kernel design where interrupt handlers reside in the kernel, interrupt handlers in Apertos are treated in a manner similar to user objects, with the exception that MetaCore will interrupt the currently executing method to allow interrupt handler objects to perform their tasks. The following series of steps take place when an interrupt occurs, as shown in Figure 4.1.

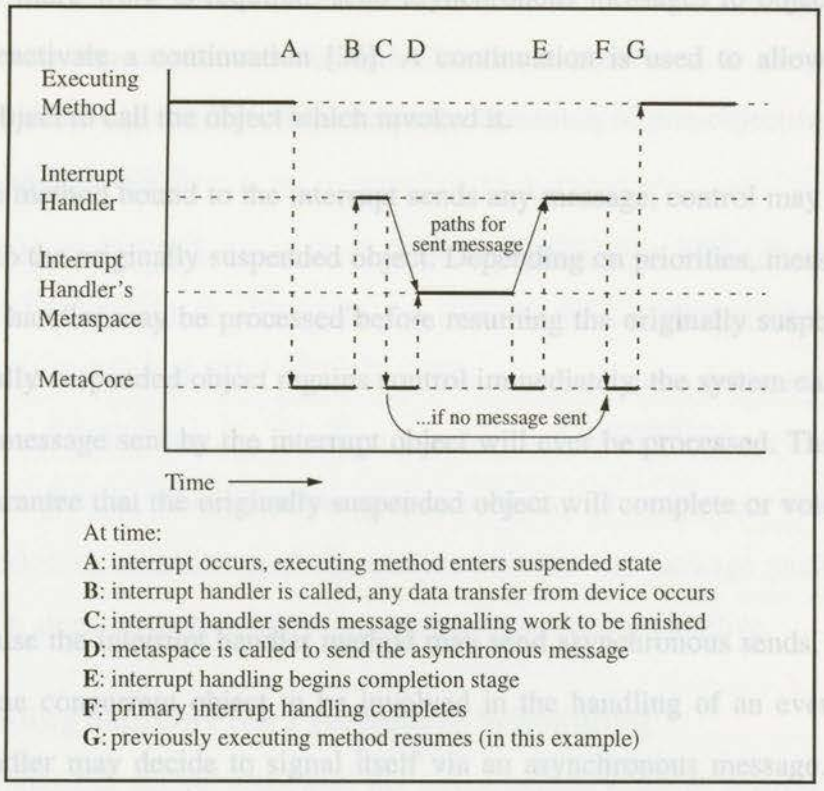


Figure 4.1 Execution Flow for a Typical Interrupt Handler

1. Control is passed directly from the currently executing object to MetaCore, analogous to a microkernel.
2. MetaCore then determines which object and which of its methods is bound to the event; if any, a message is sent to that object at the highest priority.
3. Once the interrupt handler object returns, control is returned to MetaCore, which may resume normal execution of the interrupted method.

Note that this interrupt handling method is not intended to perform any computational tasks associated with the interrupt. This call is only intended to:

- transfer data to/from a hardware device;
- decide whether more action regarding the interrupt is required; and

- if more work is required, send asynchronous messages to objects and/or reactivate a continuation [36]. A continuation is used to allow a target object to call the object which invoked it.

If the method bound to the interrupt sends any message, control may not immediately return to the originally suspended object. Depending on priorities, messages sent by the interrupt handler may be processed before resuming the originally suspended object. If the originally suspended object regains control immediately, the system cannot guarantee that any message sent by the interrupt object will ever be processed. This is because it cannot guarantee that the originally suspended object will complete or voluntarily give up the CPU.

Because the interrupt handler method may send asynchronous sends, it allows for more than one concurrent object to be involved in the handling of an event. Also, the interrupt handler may decide to signal itself via an asynchronous message. This facilitates the partitioning of an object into two parts: one which deals only with the device and one which communicates with other objects in the system [36]. This division corresponds to the high- and low-level portions of an interrupt handler in a traditional operating system [36].

The current scheduling policy is simply to let the next or current method execute. Unless a message with a higher priority is introduced into the schedule by the interrupt handler's method, any suspended object is still at the head of the schedule queue, and will always immediately regain control after handling the interrupt.

4.2 Preemptive Scheduling

Preemptive scheduling is introduced to improve system response time and exception handling. As noted in the previous section, the system may not be able to guarantee its overall performance if a faulty object is executed. An object which never returns or

yields the processor can essentially halt the system. This behavior is not acceptable for a real-time system. Preemptive scheduling is the means introduced to prevent this from occurring, by allowing a system to preempt the execution of one object in favor of the execution of another object.

Possibly the biggest hurdle in introducing preemptive scheduling to Apertos, however, is the performance penalty. In previous versions of the operating system, all metaspaces had to be notified of any preemption to ensure status information remained valid, the reason being that any interrupt handler method may use services that other objects provide [38]. In a large Apertos system, notifying all metaspaces would be a very expensive operation, considering all the context switches and message passing required. This operation alone made preemptive scheduling in Apertos prohibitive.

4.2.1 Providing Support for Preemptive Scheduling

The following section overviews the enhancements made to Apertos to introduce preemptive scheduling. The changes were made to the R3000 implementation of Apertos, but should map easily to the i486 version.

4.2.1.1 Introduction of a Real-Time Clock

To provide signals to the preemptive scheduler, a Clock object is introduced into Apertos. The Apertos Clock object to be introduced is discussed in detail in [39] and [19]. However, rather than busy-waiting, as is discussed in [39] and [19], Clock is attached to an interrupt. The interface of the Clock object introduced is given in Table 4.1. See Appendix D for exact implementation details. The AlarmInfo structure specifies information such as start time, deadline, and periodicity of the alarm.

4.2.1.2 Interrupt Handler Object

The Clock object discussed in [39] and [19] uses busy waiting to find the elapsed alarm time. This is extremely inefficient and should be replaced with a timer interrupt

mechanism. To do this, an Interrupt Event object is re-introduced from an older version of Apertos, which is bound to the R3000's generic interrupts. The task of this object is to decide if any object should be notified in the event of an R3000 generic interrupt occurring. This task is different from the responsibilities of MetaCore, since MetaCore determines the type of interrupt, i.e. if it is generic or otherwise (such as the TLB fetch exception on the R3000). At this point, the Interrupt Event object simply notifies the Clock object if the interrupt was a clock interrupt.

Method	Description
`Boolean Reset(granularity)`	changes the granularity of Clock
`Boolean ResetTime(Time, granularity)`	changes system time, granularity of Clock
`void CheckAlarms()`	force alarms to be checked (done automatically with interrupt signal)
`Time GetCurrentTime()`	returns current real time
`int NumArmedAlarms()`	number of alarms being monitored
`int NumAlarms()`	total number of alarms attached to Clock
`Alarm BindAlarm(AlarmInfo, SID, selector, parameter)`	binds a new alarm to Clock. AlarmInfo specifies real-time information; SID, selector, parameter are used to communicate with an object when the alarm expires
`Boolean AdjustAlarm(Alarm, AlarmInfo, SID, selector, parameter)`	changes information on a valid alarm
`Boolean UnbindAlarm(Alarm)`	deletes an alarm from Clock
`AlarmStatus GetAlarmInfo (Alarm)`	returns status information on a valid alarm

Table 4.1 Clock Object Interface

4.2.1.3 Attaching Scheduling Decisions to Clock Object Events

For scheduling purposes, the scheduler object requests the Clock to deliver to it a periodic alarm by calling Clock's `BindAlarm()` method at system start-up. Because of the system state at that time (system identifiers for active objects have not been initialized), the scheduler must call the method `BindAlarm()` directly rather than communicating via standard MCore message primitives. However, once the Clock is ready to

signal the scheduler, system identifiers have been initialized, so MCore's message passing primitives can and actually are used. This communication increases the overhead incurred in handling the interrupt, but must be done to remain consistent with the system and its object-oriented design.

The scheduler object also communicates directly with the Exec object, which is also on MCore. The Exec object maintains the central, root scheduler for Apertos.

4.2.2 Integrating Objects into the System

Adding user or system objects so the operating system will recognize and incorporate them involves specific steps in Apertos. Here, we are more interested in adding system objects.

Since system objects do not use reflectors such as MMiniMeta or MCOOP, which are available to application objects, system objects must be written in C++, rather than MC++. Therefore, the object is written as any ordinary passive C++ object, and then C stub routines are written and added to act as entry points to the object. Parameter lists are packaged in message structures. (The purpose of MC++ is to automate the generation of the C stub routines and message structures, but it only performs this generation for user reflectors.)

Because MDrive, the device driver reflector, currently does not exist for the R3000 version of Apertos used, all system objects, such as the Clock object, are placed directly on MCore. In `MCore::Prologue()`, an image of the object is allocated normally, as any passive object would be. MCore then proceeds to add to it properties such as a Context object, a stack, etc. to allow it to be treated by the system as an active object. The C entry stubs are used to create a table of entry points, similar to a C++ virtual function table, or a library function table. An Activity object is created, and various attributes defining interrupt masks, etc. are also set. If the object is to be accessible by

other objects, such as Clock, MZero must be notified of the new object. See Appendix C for more details on the introduction of active objects to Apertos.

4.2.3 Preemptive Scheduling Policy

At this point, we have to show how objects are actually being preemptively scheduled, and how this affects performance. The scheduling policy introduced is quite simple. As discussed earlier, the Exec structure manages a simple prioritized scheduling queue. The existing non-preemptive scheduling policy simply executes the object with the highest priority. By removing the suspended object (it is suspended at the occurrence of the timer interrupt that led to the execution of the scheduler object) and placing it back onto the queue, basic prioritized round-robin scheduling is performed instead of the given FIFO algorithm. At present, this modification is sufficient.

4.2.4 Preemption in Previous Versions of Apertos

Preemptive scheduling in Apertos has been attempted previously. But because of the performance penalty, it was dropped in favor of non-preemptive scheduling [38]. The performance penalty arose from the underlying design of Apertos at the time, requiring that all metaspaces in the system be notified during preemption.

The current design of Apertos changes this with the introduction of the Activity object, so that no notification is required. Activity objects can be seen by MetaCore, and as soon as an interrupt occurs, MetaCore saves processor state information and places the object in the `SUSPEND` state. MetaCore need not know the location of the Activity object in the meta-hierarchy; the active object's reflector is implicitly notified of the change in the object's state when MetaCore changes it.

4.2.5 Other Modifications Required

The previous sections have outlined how preemptive scheduling has been introduced to Apertos. However, to test and to use preemptive scheduling, additional changes were required.

4.2.5.1 Changes to MMiniMeta

The implementation of Apertos used in this project does not allow for inter-metaspace communication. Therefore, all test objects had to be executed on one reflector. MMiniMeta was chosen over MCOOP because of its simplicity, and because MCOOP provided no advantages over MMiniMeta for this test.

MMiniMeta maintains a queue of objects ready for execution. It will place only one object at a time into Exec's execution queue. For the existing cooperatively scheduled system, this was adequate. However, since we are limited to one reflector for user objects (because there is no inter-metaspace communication), there could only be exactly one user object in Exec's run queue at any given time. MMiniMeta has been modified to place any and all objects ready for execution into Exec's execution queue.

This solution worked satisfactorily until more than one object tried to communicate with its reflector, MMiniMeta. MMiniMeta uses a local variable, `activeObject`, to identify the currently executing object on this reflector. When an object communicates with its reflector, it doesn't identify itself; but leaves resolution of its identity to the reflector. Before the changes, only one object supported by MMiniMeta could be placed in Exec at a time, so this worked. When more than one object supported by MMiniMeta is placed in Exec, however, MMiniMeta's mechanism to identify the currently executing object can become invalid. This is one of the reasons why in previous attempts to add preemptive scheduling; all metaspaces in the system were notified of the change in the object being scheduled.

There are two ways to solve this:

- notify metaspace(s) of any change in the identity of the currently executing object; or
- provide a means to let a reflector identify the object communicating with it.

The first choice, as mentioned, has been tried [38]. However, where previous attempts notified all metaspaces, the notification could now be limited to the metaspace of the object about to be scheduled. This technique, however, will still affect the performance of any preemptive solution, and consequently the second solution was chosen.

When an object communicates with its reflector, the object's identifier is tagged onto the message during the intermediate step in MetaCore. Conceptually, the meta-meta-level is informing the meta-level of the object communicating with it. The Context structure associated with an active object can be used to uniquely identify that object. Therefore, the Context is used to update MMiniMeta's `activeObject` variable whenever an object communicates with its reflector. This scheme ensures that the `activeObject` variable is always up to date, but introduces equal overhead for all other communication in the system (such as MZero sending a message to MCore), since the modification was made to MetaCore, which is involved in all communication.

4.2.5.2 Changes to SCC and InStream

SCC is the serial device driver object, and is used for all communication on the R3000 implementation of Apertos at this time. SCC busy-waits for input regardless of whether interrupts or polling is used with the device. Even with preemptive scheduling, there can only be one active thread of control in any concurrent object at a time. Therefore, while SCC is waiting for a new input character to return, all outputs are left pending.

SCC's code was modified such that SCC will return 0 if no character is received within a short period of time. As a result, outputs generated by other objects in the sys-

tem will not remain blocked when any object requests input.

All interaction with SCC is done through the objects InStream and OutStream, which also provide the necessary buffering. InStream also had to be modified to match the changes in SCC, so now InStream will continuously poll SCC instead of SCC remaining in a poll loop for the next input character.

Ideally, SCC should be strictly interrupt driven, signalling InStream of the arrival of a new character. The implementation of this scheme is left as future work.

4.3 Integrating Real-time Constraints

4.3.1 Real-time Scheduling

Real-time systems are expected to perform their jobs within certain deadlines. It is possible to build a system whose real-time constraints can be guaranteed under specified and predictable conditions [11, 40]. In areas such as telecommunications, avionics, and process control systems, to name a few, the guarantee of real-time constraints is critical. Such systems are said to have *hard real-time constraints*. Systems which specify timing constraints as goals, where the guarantee of successful completion is desirable but not critical, are regarded as *soft real-time systems*. Soft real-time constraints require:

- the specification of the schedule (earliest possible starting) time and deadlines;
- the specification of the cycle time if the job is periodic; and
- an outline of alternative actions in case a deadline is missed [39].

4.3.2 Reflection in Real-time Scheduling

Many aspects of soft real-time programming can be viewed as reflective, or meta-level programming. Specifying soft real-time constraints changes the program's behavior in order to satisfy the specified constraints [39]. This is a case of *declarative reflection.*

Procedural reflection can be viewed as specifying the behavior for a missed deadline [8]. If performing the job is considered the base-level computation, then defining timing constraints and taking appropriate actions when a deadline is missed can be considered to be a meta-level computation.

With this division of tasks between the base- and meta-levels, the semantics of base-level programs with soft real-time constraints are exactly the same as the semantics of objects without soft real-time constraints, assuming no violation of the constraints [39]. Therefore, the meta-level involves itself with the timing constraints while the base-level only performs the required job computations, with or without timing constraints. Consequently, reflection is a powerful tool since it allows the system itself to examine the computation of its own jobs and provide the chance to adapt according to past experiences. Reflection also provides the programmer with a clear, consistent interface and design philosophy, thus aiding program development. If the meta-level is to be able to examine a real-time object's computation and adjust accordingly, then computation at the base- and meta-levels must proceed concurrently. Preemptive scheduling is the first step towards achieving this concurrency, and therefore the first step to satisfying real-time constraints in Apertos.

Within the Apertos operating system, reflection and meta-computations are used extensively, making it useful for real-time applications. However, the existence of a meta-hierarchy introduces new problems not yet thoroughly explored. The next section discusses the introduction of hierarchical scheduling to address many of these problems, and to facilitate the introduction of real-time scheduling.

4.4 Hierarchical Scheduling

Apertos inherently has a hierarchical structure [11], defined by its use of reflectors and metaspaces supporting objects. Metaspaces are themselves composed of objects which must have their own metaspaces. Therefore, a tree of meta-hierarchies expands from the root, comprising the MCore and MZero metaspaces, and terminating at user objects.

4.4.1 Hierarchical Scheduling as a Solution

By the definition of a metaspace, each object should be scheduled optimally by its metaspace, but in cooperation with other objects in the system. Note that this is not a guarantee of an object's schedulability, as each object can be made up of many jobs that have to be scheduled with the other objects in the system.

Since objects may have dissimilar scheduling requirements, one system-wide scheduler may not be sufficient, as it is possible for different metaspaces on the same machine to require different scheduling algorithms. These metaspaces, in turn, must finally be scheduled by some master scheduler. This dependency defines a hierarchy between schedulers.

4.4.2 Fitting the Apertos Paradigm

The way in which hierarchical scheduling fits into the object/metaobject paradigm and meta-computation is important in terms of its use in Apertos. To address this, we must examine the definition for meta-computing. One way to consider it is as "computing about computing", i.e. computations used to examine the environment and conditions of a job. Using this definition, hierarchical scheduling fits the paradigm in that it is "computing about scheduling". At each level, the system computes the schedules of the dependent schedules and objects.

In order to follow the paradigms used in Apertos, each object must be scheduled

based not only on its own requirements, but also in cooperation with other objects in the system. Therefore, any proposed underlying scheduling mechanism must be able to support dissimilar scheduling algorithms and requirements [19]. Thus, another objective here is to develop and refine an environment which allows different scheduling policies, whether real-time or non-real-time, to coexist on the same CPU.

4.4.3 Previous Work

A hierarchical scheduling system was introduced in the PM operating system, which is outlined in Section 2.2.4. The scheduling system is discussed in detail in [25]. The motivation behind the PM project coincides with that of this thesis, in that it tries to extend the notion of threads and processes found in traditional operating systems, and to give an application control over the scheduling of its internal threads. Highlights of PM's concepts are:

- A scheduler inside an application provides for user-level scheduling, independent of the system scheduler.
- A separate scheduling strategy for a group of threads inside an application can be very useful. Therefore, another scheduler is needed below the application level scheduler.
- The problem of uneven distribution of computing time can be avoided for applications that create a large number of threads.
- The relation of a thread to its application matches nicely to that of an object to its metaspace.

Work experimenting with schedulers in user space has been done using BirliX [30]. The aim was to support the implementation and measurement of distinct scheduling strategies, the classification of applications, the investigation of centralized and distributed scheduling, and investigations of interfaces between scheduling distinct resources.

A hierarchy defined by application-specific schedulers placed in user space, supported by a system-specific scheduler, was chosen.

Chorus [20, 21, 22], Amadeus [28], and Oisin [31] all use clusters to group objects with similar requirements in order to address poor performance. This creates a hierarchy between an object, its cluster, and the system, so the system never manages objects directly. Presently, Apertos manages all active objects in the system at the lowest level. By modifying this, we hope to achieve the same kind of performance gains realized by introducing clusters to these systems.

4.4.4 The Model

In this section, we present a recursive model for defining hierarchical scheduling.

4.4.4.1 List of Terms

I	number of meta-levels in the system
M_i	a metaspace at meta-level i, $0 \le i < I$ (the root metaspace is defined as M_0; no metaspaces exist at level I, only objects)
D_i^m	number of metaspaces directly dependent on M_i
D_i^o	number of objects directly dependent on M_i
D_i	$D_i^m + D_i^o$
DS_i^m	set of schedules for metaspaces directly dependent on M_i
DS_i^o	set of schedules for objects directly dependent on M_i
S_{i+1}	$DS_i^m \cup DS_i^o$
N_i	number of jobs in metaspace M_i
$\{J_{i,0}, \ldots, J_{i,N_i-1}\}$	set of jobs in metaspace M_i
JS_i	set of schedules for jobs in metaspace M_i
S_i	schedule of metaspace M_i ; $S_i = JS_i \cup S_{i+1}$
$M_{i+1,j}$	metaspace j which is directly dependent on metaspace M_i , $0 \le j < D_i^m$
$O_{i+1,j}$	object j which is directly dependent on metaspace M_i , $0 \le j < D_i^o$
$S_{i+1,j}$	schedule j of a dependent metaspace or object, $0 \le j < D_i$

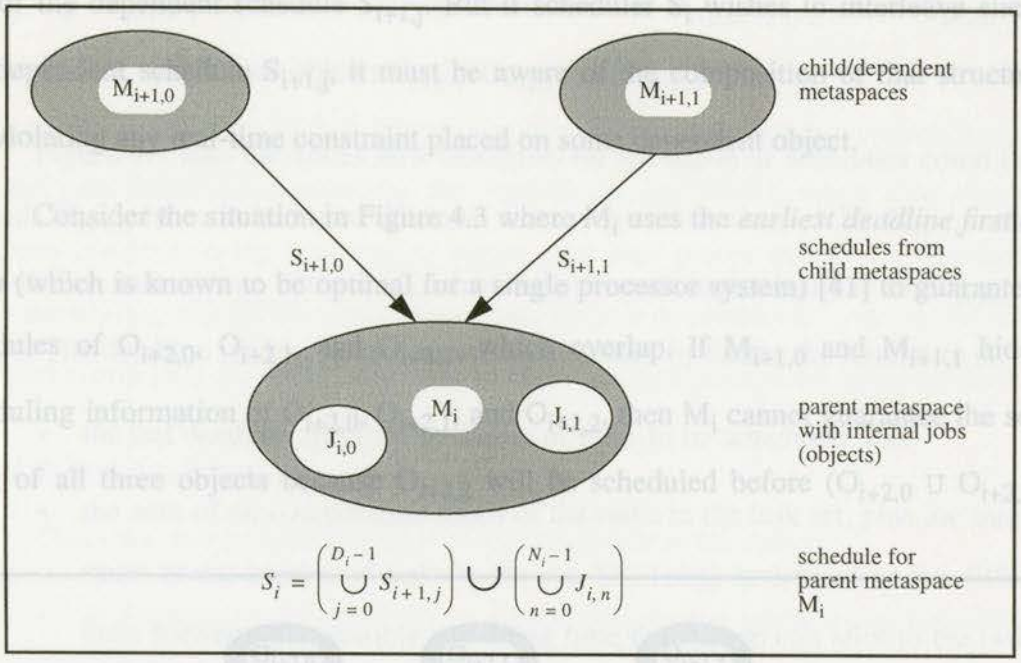


Figure 4.2 Definition of Hierarchical Scheduling

4.4.4.2 Model Definition

See Figure 4.2. Using the existing meta-hierarchy as a basis, it can be stated that:

- scheduler S_i = {all dependent $S_{i+1,j}$, all jobs $J_{i,n}$ within that metaspace};
- job = active object to be scheduled within a given metaspace (that is, a metaobject to level i+1).

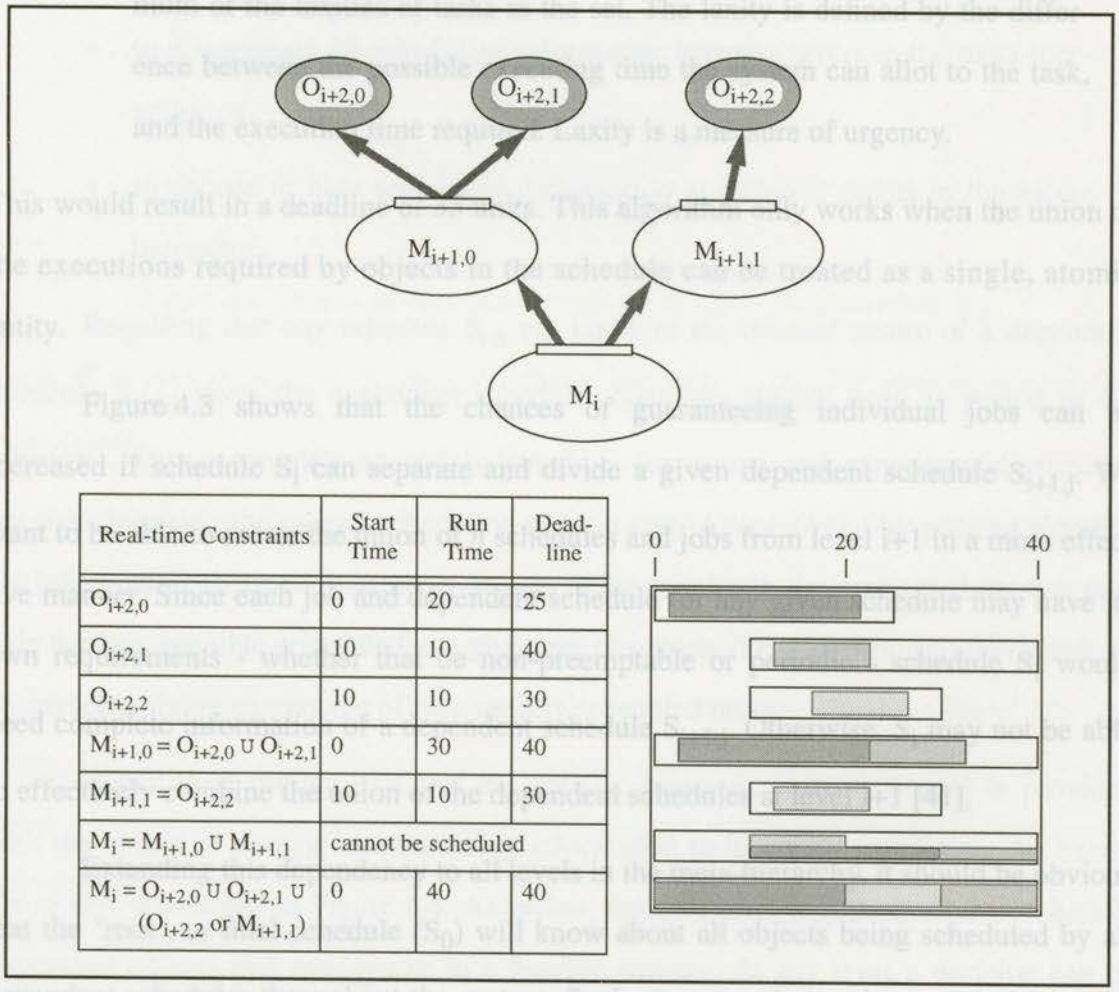
I.e., the given schedule at any level must combine the schedules of all objects within that metaspace, and the schedules of all dependent metaspaces and objects. An object at level i+1 defines a base-level computation, and the i'th level represents its metaspace. It is assumed that every object is dependent upon exactly one metaspace.

4.4.4.3 A Non-optimal Solution

If we assume that the time-slice for a given schedule $S_{i+1,j}$ is disjoint from all other schedules at level i+1, then schedule S_i need not be concerned with the composi-

tion of the dependent schedule $S_{i+1,j}$. But if scheduler S_i wishes to interleave slices of any dependent schedule $S_{i+1,j}$, it must be aware of the composition of that structure or risk violating any real-time constraint placed on some dependent object.

Consider the situation in Figure 4.3 where M_i uses the *earliest deadline first algorithm* (which is known to be optimal for a single processor system) [41] to guarantee the schedules of $O_{i+2,0}$, $O_{i+2,1}$, and $O_{i+2,2}$, which overlap. If $M_{i+1,0}$ and $M_{i+1,1}$ hide the scheduling information of $O_{i+2,0}$, $O_{i+2,1}$, and $O_{i+2,2}$, then M_i cannot guarantee the scheduling of all three objects because $O_{i+2,2}$ will be scheduled before $(O_{i+2,0} \cup O_{i+2,1})$. If



Real-time Constraints	Start Time	Run Time	Dead-line
$O_{i+2,0}$	0	20	25
$O_{i+2,1}$	10	10	40
$O_{i+2,2}$	10	10	30
$M_{i+1,0} = O_{i+2,0} \cup O_{i+2,1}$	0	30	40
$M_{i+1,1} = O_{i+2,2}$	10	10	30
$M_i = M_{i+1,0} \cup M_{i+1,1}$	cannot be scheduled		
$M_i = O_{i+2,0} \cup O_{i+2,1} \cup (O_{i+2,2}$ or $M_{i+1,1})$	0	40	40

Figure 4.3 2 Dependent Metaspaces Managing 3 Objects

$M_{i+1,0}$ and $M_{i+1,1}$ pass the scheduling information of $O_{i+2,0}$, $O_{i+2,1}$, and $O_{i+2,2}$ to M_i, then M_i is able to schedule objects $O_{i+2,0}$, $O_{i+2,1}$, and $O_{i+2,2}$.

Figure 4.3 also illustrates how deadlines for the union of schedules could be calculated. $M_{i+1,0}$ has a deadline of 40 units, and an execution time of 30. $M_{i+1,0}$ could be scheduled to start at time interval 10, which would result in the deadline for $O_{i+2,0}$ being missed. The deadline for $M_{i+1,0}$ could also be calculated by taking the minimum of:

- the last deadline specified in the set of tasks to be scheduled, and
- the sum of all computation times of the tasks in the task set, plus the minimum of the laxities of tasks in the set. The laxity is defined by the difference between the possible executing time the system can allot to the task, and the execution time required. Laxity is a measure of urgency.

This would result in a deadline of 35 units. This algorithm only works when the union of the executions required by objects in the schedule can be treated as a single, atomic entity.

Figure 4.3 shows that the chances of guaranteeing individual jobs can be increased if schedule S_i can separate and divide a given dependent schedule $S_{i+1,j}$. We want to be able to create the union of *n* schedules and jobs from level i+1 in a more effective manner. Since each job and dependent schedule for any given schedule may have its own requirements - whether that be non-preemptable or periodic - schedule S_i would need complete information of a dependent schedule $S_{i+1,j}$. Otherwise, S_i may not be able to effectively combine the union of the dependent schedules at level i+1 [41].

Extending this dependency to all levels in the meta-hierarchy, it should be obvious that the 'root', or final schedule (S_0) will know about all objects being scheduled by all dependent schedules throughout the system. Such an extrapolation, however, negates the advantages of a hierarchical structure, specifically not requiring that all objects be sched-

uled using the same algorithm. This technique, however, is proposed for Apertos in [6], where policy objects that define scheduling policies are separated into intended policy objects and a base policy object. Intended policy objects comprise the hierarchical structure and pass information regarding the importance, periodicity, worst case processing time, and deadline to the base policy object. The base policy object is responsible for which scheduling algorithms - rate monotonic, earliest deadline, etc. - to use. An adaptation mechanism [42] would be integrated so that a common algorithm could be used.

4.4.4.4 A Feasible Solution

There are two options available to any scheduler in the system:

- to disseminate all scheduling information between layers in the meta-hierarchy; or
- to choose to hide scheduling information at different points in the meta-hierarchy.

Requiring that any schedule $S_{i,n}$ not know of the internal nature of a dependent schedule $S_{i+1,j}$ gives the dependent scheduler complete control over its period in the schedule. Having complete control is beneficial for testing and experimental purposes but risks lowering the overall scheduling success rate. Alternatively, by passing scheduling information about all concurrent objects to the root level, the root scheduler can provide the best possible schedule for a given set of objects. This, in turn, should be able to guarantee a higher percentage of successfully scheduled tasks.

Rather than forcing either of the options, however, each metaspace is provided with the option of passing on scheduling information to its parent metaspace. An example of this is outlined in Figure 4.4. As before, the root level is still allowed to schedule dependent objects and metaspaces in a feasible manner. At any level a decision can be made as to whether to hide or pass on all scheduling information. While there still exists a

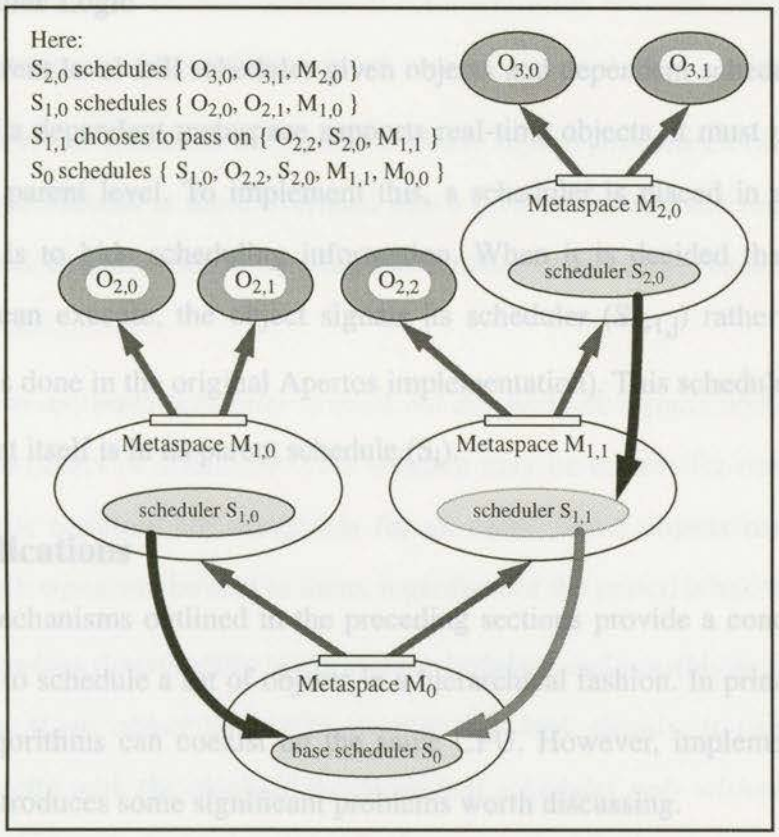


Figure 4.4 Proposed Real-time Scheduling Policy in the Meta-hierarchy

trade-off in performance vs. privacy, the choice becomes application specific and tunable to an extent.

Even though there may not be an optimal solution available for a given hierarchical schedule, there are still benefits to a hierarchical structure for scheduling. It allows different objects and metaspaces to be scheduled with different policies, whether it be earliest deadline first [41], or least laxity [41], or any new experimental scheduling algorithm. The fact that an optimal solution may not exist, however, is not of prime importance, as we wish to consider a dynamic environment where pre-run-time scheduling is not possible.

4.4.4.5 Scheduler Logic

The parent level still schedules given objects and dependent schedules in a feasible manner. If a dependent metaspace supports real-time objects, it must pass this information to the parent level. To implement this, a scheduler is placed in any metaspace whose policy is to hide scheduling information. When it is decided that a dependent active object can execute, the object signals its scheduler ($S_{i+1,j}$) rather than the root scheduler (as is done in the original Apertos implementation). This scheduler ($S_{i+1,j}$) must then ensure that itself is in its parent schedule (S_i).

4.4.5 Complications

The mechanisms outlined in the preceding sections provide a concise methodology by which to schedule a set of objects in a hierarchical fashion. In principle, differing scheduling algorithms can coexist on the same CPU. However, implementing such an architecture introduces some significant problems worth discussing.

4.4.5.1 Time-Dependent Signals to Scheduler Objects

It is reasonable to assume that a time-dependent scheduler may wish to use preemptive scheduling to schedule the objects and the dependent schedules it manages. To implement this strategy, the scheduler may wish to receive signals from the system clock, as described in 4.2.1.1. However, there is a problem in that the clock will continue to send signals to that time-dependent scheduler even when another scheduler has taken control. For example, should a given scheduler be given half of the available schedule time by its meta-scheduler, then half of the clock signals will then be unnecessary and invalid. The problem can be handled in several ways:

- *add functionality to the clock so that it only emits periodic signals within a certain period.* Since it is the scheduler that requests periodic signals from the clock, and its meta-scheduler decides upon the period during which the

scheduler can execute, additional communication between a scheduler and its meta-scheduler will be needed.

- *ensure the scheduler only uses the timely signals from the clock.* This technique introduces more overhead into a given scheduler and requires additional communication between the scheduler, its meta-scheduler, and the clock.

- *allow any meta-scheduler to mask out any periodic signals sent to a dependent object or scheduler.* This solution may be correct for one scheduler but is certainly not acceptable for all cases. Some objects may want all clock signals to be sent to them, regardless of the period it has to execute.

- *introduce functionality to the meta-schedulers and possibly to the clock so that if an object wishes to receive all clock signals, it communicates directly with the clock. If the dependent scheduler only wishes to receive clock signals during its period in the meta-schedule, it requests this from its meta-scheduler.* Depending on implementation, the meta-scheduler can either make this request to the clock on behalf of the scheduler and mask out unneeded requests, or add functionality to the Clock to send signals only within the period specified by the meta-scheduler. This solution was chosen since it is the most flexible and incurs the least overhead.

4.4.5.2 Suspension of an Executing Object

Without any hierarchical structuring, preemptive scheduling is easy to implement, since the task being preempted is simply added back into the system (root) schedule, to resume execution at a later time. However, with the introduction of hierarchical scheduling, where the root scheduler may not know of all the objects to be scheduled; it is incorrect to simply add the suspended object to the root schedule for re-execution. Therefore, the algorithm has to be modified so that:

- if a scheduler passes scheduling information on to its meta-scheduler, it only acts as a reference to its meta-scheduler and passes *all* information through, simply acting as a router of messages.
- if a scheduler maintains its own execution queue, it has to request to its metaspace to add itself to its meta-schedule when an object it supports asks to be scheduled.
- a common portion of the structure containing state information for any given object can be shared between all schedulers. This includes the state of the object and identifies the object's scheduler. A scheduler should not need to be aware of a given object to have access to this information.
- when a given scheduler decides to preempt the currently running object, it should place that object into the `SUSPEND` state, thus implicitly notifying that object's scheduler of the state change.
- the scheduler does not immediately add the object just suspended to its execution queue. Instead, it checks that object's state information to identify that object's scheduler. If the scheduler specified is the same as itself, the object can be added to the scheduler's execution queue. If it is different, the object's scheduler is also placed in the `SUSPEND` state. This is done by sending an asynchronous message notifying the suspended scheduler that an object it manages has been suspended. This step is repeated to trace down the hierarchy to the active scheduler. Therefore, if scheduler S_3 preempts an object scheduled by S_1, which in turn is scheduled by S_2, which is scheduled by S_3, then S_3 places objects S_1 and S_2 into the `SUSPEND` state and adds S_2 to its schedule. S_3 can then allow some other dependent object or metaspace to execute.

4.4.5.3 Root Scheduling

All metaspaces and schedulers are allowed to use different algorithms in order to provide the services required for the objects and metaspaces they support. In theory, for any object there could be a metaspace, either existing or constructed, to provide the services required. This flexibility introduces a considerable problem for the root scheduler, which must somehow manage possibly disjoint and dissimilar scheduling algorithms.

A policy-free scheduling technique, such as that employed by RT-Mach [34, 43], was chosen to address this issue. The base scheduler simply provides time 'slots' which can be allocated to dependent metaspaces and objects. When an object (or a scheduler) requests to be scheduled, it specifies the periodicity (if any), worst case execution time, etc. The base scheduler then attempts to add it to the schedule in a feasible manner. If the object cannot be added to the existing schedule, the scheduler which decided this should communicate this information to the requesting object so that alternative actions can be taken.

Any object requesting to be scheduled can make no assumptions as to how it will be scheduled. It only knows that it will be scheduled if the request is successful. It is up to the schedulers in the metaspaces that are dependent on MCore to guarantee the implementation of algorithms, such as least laxity real-time scheduling, etc. For example, scheduler $S_{1,0}$ in Figure 4.4 would perform its own scheduling. If no particular scheduling algorithm is required by objects, as in $S_{1,1}$ in Figure 4.4, then information about the objects to be scheduled is passed to the root scheduler.

4.4.6 Implementation of Hierarchical Scheduling

To implement hierarchical scheduling in Apertos, modifications were limited to MMiniMeta. It is difficult to fully test hierarchical scheduling since MZero no longer supports multiple dependent reflectors, and since no user reflectors can receive messages originating from their meta-layers. For testing purposes, only communication between

application objects on the same metaspace was supported, as was used in the Apertos benchmark. Instead of MMiniMeta requesting MCore's Exec object to schedule, then later execute a given object, MMiniMeta immediately executes the target object to which the synchronous call or reply is sent.

Chapter 5

Performance Measurement and Analysis

This chapter describes how performance measurements were made. After outlining the tests, we show the effect of our preemptive and hierarchical scheduling schemes in improving the real-time performance of Apertos.

5.1 Test Case

A simple test was constructed to verify the correct operation of round-robin preemptive scheduling. An object, Test1, asynchronously sends messages to start two other objects, Test2 and Test3, both of which once started, will run indefinitely. After starting Test2 and Test3, Test1 produces some output and completes. Test1 and Test2 can both

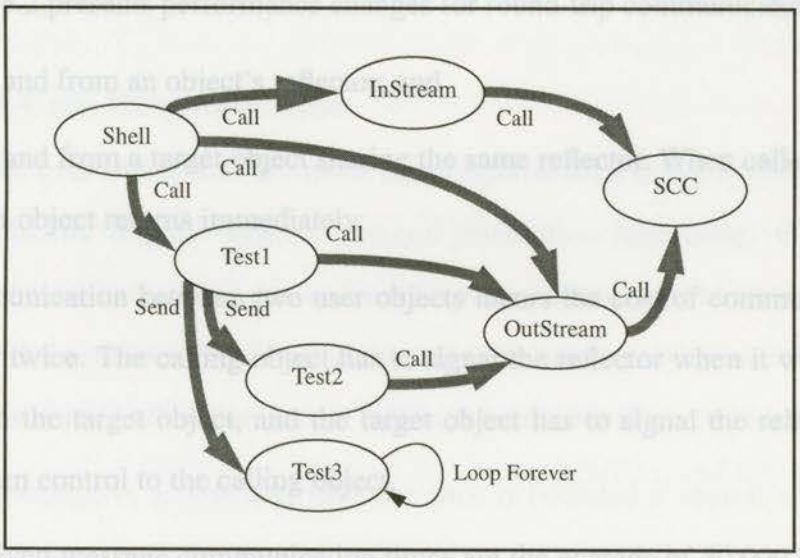


Figure 5.1 Communication Diagram of Test Case

send output to OutStream, and finally to SCC. Test3 simply sits in a loop, never completing or communicating with any other object. Shell, which already exists in Apertos, is used to start Test1 by issuing a synchronous call. Shell can also provide output and request input. See Figure 5.1.

Once this test has started, Test3 will either be in the run queue or currently executing. Test2 is continuously communicating with OutStream. Shell is either waiting for input, waiting for Test1 to complete, or producing output.

Test1 was also used for timing communication performance. To time round-trip communication with an object's reflector, an extra method was added to MMiniMeta, which simply returns immediately to the application object by invoking MetaCore's R primitive. To time communication performance with another application object, a method was introduced to Test2, which also simply returns immediately. MMiniMeta's `Call()` and `Reply()` methods were used for timing communication between Test1 and Test2.

5.2 Performance Results

Table 5.2 presents performance changes for round-trip communication times for:

- to and from an object's reflector; and
- to and from a target object sharing the same reflector. When called, the target object returns immediately.

Communication between two user objects incurs the cost of communicating with their reflector twice. The calling object has to signal the reflector when it wishes to communicate with the target object, and the target object has to signal the reflector when it wishes to return control to the calling object.

The given message communication times are the average of 50,000 iterations. All

Measurement (in μsec)	Communication time between an object and its reflector	Communication time between 2 user objects
Standard Apertos (benchmark)	71	872
Preemptive Scheduling disabled	102	1075
Preemptive Scheduling	113	1157
Hierarchical Scheduling (no interrupts)	80	179
Hierarchical Scheduling	91	198

Table 5.1 Comparison of Communication Times

measurements are from using the timing board in a Sony computer, based on the 25Mhz MIPS R3000 processor. The timing board is accurate to within one microsecond. Version 0.5.1 implementation of Apertos, with default system objects, was used both for the benchmark for all tests, and for a basis for development.

5.2.1 Performance of the Test Case

Without preemptive scheduling, the test case simply halts as soon as Test3 gains control, since it never releases it. With preemptive scheduling, the test case will execute indefinitely, as it should. Hierarchical scheduling continues to use preemptive scheduling, so the test case executes as expected.

5.2.2 Worst Case Response Time

The test case shows that the worst case response time in the system has changed significantly. In the original system, without preemptive scheduling, the worst case response time was infinite. This was because no other object could execute until the running object at the time had completed. If the currently running object never completes or yields, no other objects other than simple interrupt handlers will ever execute.

With preemptive scheduling, response time is bounded if objects share the same priority. Worst case response time, therefore, becomes equal to the number of tasks

ahead of it in the run queue multiplied by the timeslice, plus the time spent in interrupt handlers and in making scheduling decisions. If there exists a task in the run queue with a higher priority, response time becomes immeasurable. However, this is *only* due to current scheduling policy. If scheduling policy is modified such that each priority queue is guaranteed to receive a portion of the schedule time, then response time can again become measurable.

Hierarchical scheduling improves performance by improving communication time between objects. It also facilitates the introduction of application-specific scheduling between related groups of objects, which would further improve performance.

5.2.3 Worst Case Interrupt Disable Time

The modifications to allow MMiniMeta to resolve which application object is communicating with it come with a cost of 9 microseconds, or enough time to execute approximately 24 machine instructions. This is illustrated in Table 5.2 by the difference in the communication time between an object and its reflector for the standard Apertos benchmark and hierarchical scheduling without interrupts.

The additional code introduced to MMiniMeta and the system, to perform preemptive scheduling, increases the communication time between an object and MMiniMeta by another 21 microseconds to a total of about 30 microseconds per call to the reflector. This is illustrated by the difference between the standard Apertos benchmark and when preemptive scheduling is disabled. On the test machine, this is enough time to execute approximately 80 processor instructions, assuming no wasted cycles for memory fetches [44]. Communication time between two user objects increases more because of the changes made in MetaCore which affect communication in general, including communication between MMiniMeta and MZero, and between MZero and MCore.

Hierarchical scheduling significantly improves the worst case interrupt disable time over both the standard Apertos benchmark and preemptive scheduling by removing

MZero and MCore from the work associated with application object communication. All reflectors execute with interrupts disabled, and there is no chance of handling interrupts during communication between reflectors. See Figure 5.2 and Figure 5.3 for how communication between active application objects has changed.

5.2.4 Communication Times

For preemptive scheduling, the communication times between both an object and its reflector, as well as between two application objects, have increased, as is shown in

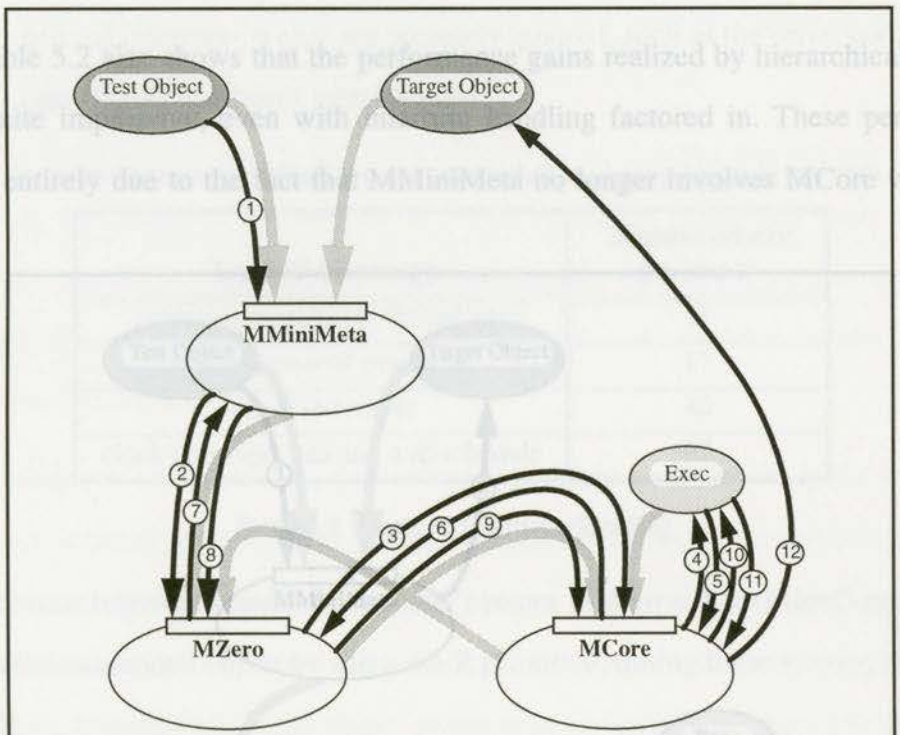


Figure 5.2 Original Control Path for Communication in Apertos

Table 5.2. This increase is attributed to modifications in MMiniMeta and the system, as discussed earlier.

The effects of interrupts are shown in the timing differences between the two implementations of preemptive scheduling for communication between two objects. The clock interrupts do not show up in the communication with a reflector because the probability of a timing interrupt occurring within that period is negligible. For communication with another user object, we are guaranteed a minimum of one, and possibly two clock interrupts during that period.

Table 5.2 also shows that the performance gains realized by hierarchical scheduling are quite impressive, even with interrupt handling factored in. These performance gains are entirely due to the fact that MMiniMeta no longer involves MCore via MZero

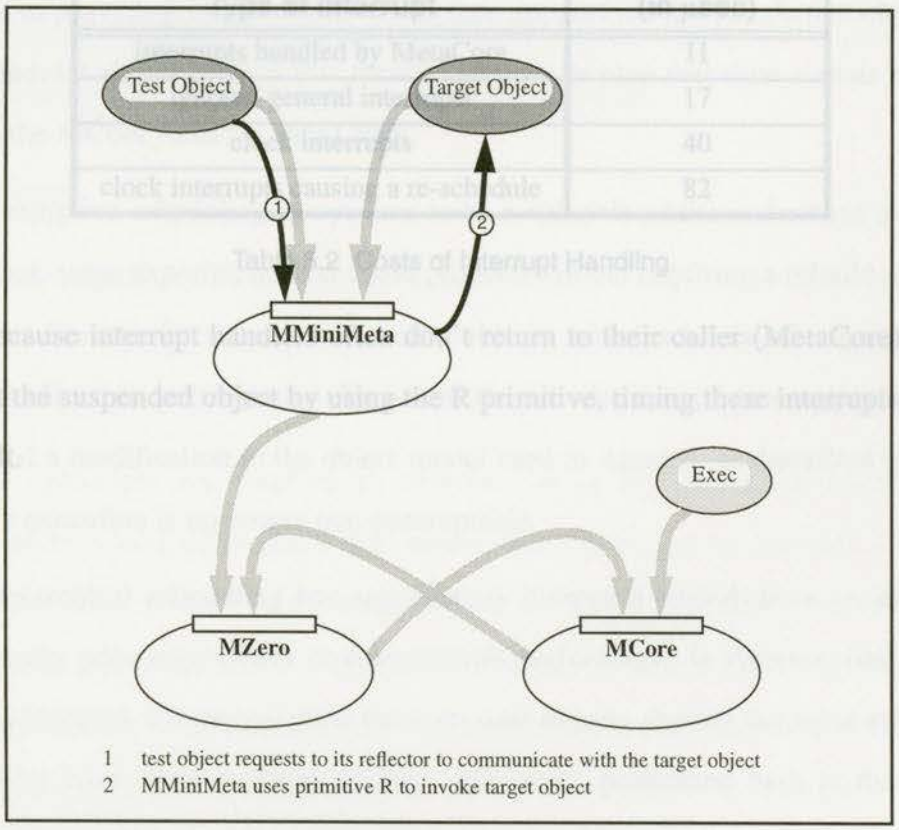


Figure 5.3 Control Path for Communication in Apertos using Hierarchical Scheduling

Chapter 6

Conclusions

The work here has shown that active objects provide a suitable model to structure an operating system, providing clarity and consistency of interfaces.

The Clock object which has been introduced to Apertos has proven to be efficient, stable, and reliable. The manner in which the Clock object has been introduced should aid in providing comprehensive real-time support in Apertos. Active objects, such as the scheduler object used in this thesis, can now receive real-time signals if they are placed on the MCore reflector with Clock.

Preemptive scheduling has proven to be a valuable addition, because even with a faulty object, some experimentation could proceed without requiring a rebuild and reboot. Apertos has become far more stable, better able to withstand a failed object without a substantial effect on performance. However, the introduction of preemptive scheduling has required a modification to the object model used in Apertos, as described in [19, 39]; an object's execution is no longer non-preemptable.

Hierarchical scheduling has significantly increased performance. A major complaint, namely poor inter-object communication performance in Apertos, has been successfully addressed. Communication between user objects sharing the same reflector has improved by more than a factor of four. The work performed here is the first step towards experimentation with various scheduling algorithms, real-time and non-real-time,

without significantly affecting the remaining system. However, hierarchical scheduling requires that MCore allow other objects in the system to directly control the execution of application objects in the system, through the use of MetaCore's primitives. Before hierarchical scheduling, all object executions began from MCore calling MetaCore's R primitive.

While studying the Apertos source code, we became more convinced that even with the object paradigm, accurate and complete documentation of source code is essential. Object interfaces must be clearly defined to avoid questions regarding subtleties and implementation details. Often, the reasoning behind design decisions would not become clear until one had the total picture of the operation of Apertos in mind; a portion of the picture is often not sufficient.

Finally, this thesis has shown that an operating system employing a meta-hierarchy and reflection can be a viable as a platform for supporting object-oriented programs. The performance is acceptable, and the system can be stable.

6.1 Future Work

Many improvements have been made during the evolution of Apertos. However, some additional changes are necessary if real-time support is to be completely implemented. This section discusses the areas where improvements should be made.

All interrupts are disabled in Apertos during all activity in reflectors. This improves performance since global information and objects can be accessed directly (not via standard paths) without introducing mutual exclusion problems. This is acceptable for the current implementation of Apertos, but is not sufficient for a real-time environment, where there is a real-time clock. Various interrupts, including clock interrupts are currently lost due to the excessive length of time that execution may take place with interrupts disabled.

Appendix B should make it apparent that there is a large amount of overhead involved in actually sending a message between two active objects in the current implementation of Apertos. In order to send a message from an active object to its reflector, passive objects are encapsulated, then re-encapsulated again in other objects. Here, the class inheritance for message classes should be revamped from three separate hierarchies referencing each other, to one, beginning with a Message class.

An immediate future work is the full implementation of message passing primitives to support inter-metaspace communication. When combined with a real-time scheduler, such primitives could provide real-time support to all active objects in the system.

References

[1] D. L. Carver. "Integrated Modeling of Distributed Object-Oriented Systems." *Journal on Systems Software*, (26):233–244, 1994.

[2] S. Graupner, W. Kalfa, and F. Schubert. "Multi-level Architecture of object-oriented Operating Systems." Technical Report TR-94-056, International Computer Science Institute, Berkeley, CA, USA, November 1994.

[3] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[4] Y. Yokote, F. Teraoka, and M. Tokoro. "A Reflective Architecture for an Object Oriented Distributed Operating Systems." In *European Conference on Object-Oriented Programming '89*, March 1989.

[5] F. J. Hauck. "PM: A Distributed Object-Oriented Operating System." In *ECOOP '93*, 1993.

[6] Y. Yokote, F. Teraoka, A. Mitsuzawa, N. Fujinami, and M. Tokoro. "The Muse Object Architecture: A New Operating System Structuring Concept." *Operating Systems Review*, 25(2), April 1991.

[7] Y. Yokote, F. Teraoka, M. Yamada, H. Tezuka, and M. Tokoro. "The Design and Implementation of the Muse Object-Oriented Distributed Operating System." In *First Conference on Technology of Object-Oriented Languages and Systems*, October 1989.

[8] P. Maes. *Meta-level Architecture and Reflection*, chapter "Issues in Computational Reflection," pages 21–35. North-Halland, 1988.

[9] M. Tokoro and A. Yonezawa, editors. *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, 1987.

[10] Y. Yokote. "Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach." In *International Symposium on Object Technologies for Advanced Software (ISOTAS '93)*, 1993.

[11] Y. Yokote. "The Apertos Reflective Operating Systems: The Concept and Its Implementation." In *Conference on Object-Oriented Programming, Languages and Applications*, 1992.

[12] U. M. Borghoff and K. Nast-Kolb. "Distributed Systems: A Comprehensive Survey." Technical Report TUM-I8909, Technische Universitaet Muenchen,

Munich, Germany, November 1989.

[13] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. "The Clouds Distributed Operating System." *IEEE Computer*, pages 34–44, November 1991.

[14] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International, Inc., 1992.

[15] V. F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois, 1991.

[16] A. Chatterjee, A. Khanna, and Y. Hung. "ES-Kit: an Object-Oriented System." *Concurrency: Practice and Experience*, 3(6):525–539, December 1991.

[17] K. Murata, R. N. Horspool, Y. Yokote, E. G. Manning, and M. Tokoro. "Cognac: a Reflective Object-Oriented Programming System using Dynamic Compilation Techniques." In *Japan Society of Software Science and Technology*, 1994.

[18] K. Murata, R. N. Horspool, E. G. Manning, Y. Yokote, and M. Tokoro. "Unification of Active and Passive Objects in an Object-Oriented Operating System." In *1995 International Workshop of Object Orientation in Operating Systems*, 1995.

[19] Y. Honda and M. Tokoro. "Object-based Concurrent Reflective Architecture for Time-Dependent Computing." Technical Report SCSL-TR-93-002, Sony Computer Science Laboratory Inc., Tokyo, Japan, March 1993.

[20] R. Lea and C. Jacquemot. "The COOL architecture and abstractions for object-oriented distributed operating systems." In *5th ACM SIGOPS European Workshop on Models and Paradigms for Distributed Systems Structuring*, Rennes, France, September 1992.

[21] Y. Yokote. "Object-Oriented Distributed Operating Systems." Technical Report SCSL-TR-92-013, Sony Computer Science Laboratory Inc., Tokyo, Japan, May 1992.

[22] P. Amaral, R. Lea, and C. Jacquemot. "Implementing a modular object oriented operating system on top of CHORUS." In *OpenForum '92*, Utrecht, The Netherlands, November 1992.

[23] A. D. Birrell and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[24] J. Kleinoder. "Object- and Memory-Management Architecture: A Concept of Open, Object-Oriented Operating Systems." In A. Bode and H. Wedekind, editors, *Parallel Computer Architectures: Theory, Hardware, Software, and Applications*, SFB Colloquium SFB 182 and SFB 342, Munich, October 8-9 1992.

[25] J. Kleinoder and T. Riechmann. "Hierarchical Schedulers in the PM System-Architecture." Technical Report TR-I4-94-16, Friedrich Alexander University, Erlangen-Nurnberg, Germany, June 1994.

[26] G. Hamilton and P. Kougiouris. "The Spring Nucleus: A MicroKernel for Objects." In *1993 Summer USENIX Conference*, June 1993.

[27] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Mandany, M. N. Nelson, M. L. Powell, and S. R. Radia. "An Overview of the Spring System." In *Compcon Spring 1994*, February 1994.

[28] V. Cahill, S. Baker, B. Tangney, C. Horn, and N. Harris. "On Object Orientation as a Paradigm for General Purpose Distributed Operating Systems." In *1992 ACM SIGOPS European Workshop*, December 1992.

[29] D. Acton, T. Coatta, and G. Neufeld. "The Raven System." Technical Report TR 92-15, UBC Computer Science Department, Vancouver, B.C., Canada, August 1992.

[30] W. Kalfa. "Proposal of an External Processor Scheduling in Micro-Kernel Based Operating Systems." Technical Report TR-92-028, International Computer Science Institute, Berkeley, CA, USA, May 1992.

[31] V. Cahill and A. Kramer. "OISIN: Operating System Support for Objects in a Distributed Environment." In *ECOOP/OOPSLA '90 Workshop on Object Orientation in Operating Systems*, 1990.

[32] K. Murray, T. Wilkinson, P. Osmon, A. Saulsbury, T. Stiemerling, and P. Kelly. "Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel." Technical Report TCU/SARC/1993/9, SARC, City University, Swedish Institute of Computer Science, Imperial College, 1993.

[33] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: a New Kernel Foundation for UNIX Development." In *Summer 1986 USENIX Conference*, pages 93–112, 1986.

[34] H. Tokuda, T. Nakajima, and P. Rao. "Real-Time Mach: Towards a Predictable Real-Time System." In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.

[35] T. Tenma. "MC++ Language Manual v0.2." Technical report, Sony Computer Science Laboratory Inc., Tokyo, Japan, March 1995.

[36] J. Itoh and Y. Yokote. "Concurrent Object-Oriented Device Driver Programming in Apertos Operating System." Technical Report SCSL-TM-94-005, Sony Computer Science Laboratory Inc., Tokyo, Japan, August 1994.

[37] N. Fujinami and Y. Yokote. "Naming and Addressing of Objects without Unique

Identifiers." In *12th International Conference on Distributed Computing Systems*, 1992.

[38] K. Murata, Y. Yokote, and M. Tokoro. "A Reflective Network System Using Concurrent Objects and Meta Architecture." Technical Report SCSL-TM-93-010, Sony Computer Science Laboratory Inc., Tokyo, Japan, July 1993.

[39] Y. Honda and M. Tokoro. "Soft Real-Time Programming through Reflection." In *IMSA '92 International Workshop on Reflection and Meta-level Architecture*, 1992.

[40] J. A. Stankovic. "Misconceptions About Real-Time Computing." *IEEE Computer*, 21(10):10–19, October 1988.

[41] M. L. Dertouzos and A. K. Mok. "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks." *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.

[42] J. A. Stankovic and K. Ramamritham. "The Spring Kernel: A New Paradigm for Real-Time Operating Systems." *Operating Systems Review*, 23(3), July 1989.

[43] T. Nakajima, T. Kitayama, and H. Tokuda. "Experiments with Real-Time Servers in Real-Time Mach." In *Proceedings of the USENEX Mach III Symposium*, pages 1–19, April 1993.

[44] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[45] P. A. Laplante. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, 1993.

[46] Y. Yokote and T. Tenma. "Writing Apertos Objects, v0.5.2, revision 2." Technical report, Sony Computer Science Laboratory Inc., Tokyo, Japan, March 1995.

Appendix A

Definitions in Apertos

A.1 Definitions Relating to the Object Paradigm

Class: A class is an entity which describes the similarity shared by a set of objects. For example, a class contains the definition of the methods which can access the internal structure of an object. A class also acts as a template for creation of an object [6].

Class Inheritance: Class inheritance is a mechanism by which a class is defined as a subclass of another class. Classes compose a hierarchical structure to represent differences and to encourage incremental programming. Inheritance is usually a programming and compile-time facility [6]. Multiple inheritance is not supported by all languages and systems.

Object: An object is an instantiation of a class, so has local storage and methods as defined by its class, and provides a uniform interface to its external environment. The object's local storage is accessed by its methods, which are executed upon receipt of incoming messages [6]. An object can be conceptually viewed as a small, autonomous computer which has local storage for computation, and is dynamically created and destroyed [4].

Passive Object: A passive object is a type of object which does not address synchronization or mutual exclusion problems. A passive object is owned, managed, and has

methods which are executed by at least one thread of control. However, when more than one thread of control is active in a passive object at a time, synchronization and mutual exclusion problems arise. Most languages, such as C++ and Smalltalk, support only passive objects in their object model.

Active Object: An active object, also known as a *concurrent object*, is an object which augments it local storage and methods with a virtual processor. Local storage of an active object is accessed only by methods executed by the virtual processor. Exactly one executing thread of control is associated with a concurrent object at any time. An active object has facilities associated with it which address synchronization problems; concurrent requests are synchronized at the entry point of the object [6].

Metaobject: A metaobject is an object which provides a portion of, or all of an executing environment, for an object. Metaobjects implement the meta-functions of objects it supports, such as scheduling, communication, and memory management [6]. As such, a metaobject can be viewed as defining the meta-computation of objects, or (a portion of) a virtual computer [4].

Metaspace: An object's metaspace is a collective union comprised of all the metaobjects that an object employs. It can be viewed as a virtual machine, or an optimized operating environment for the object [6].

Base-level Computation: Base-level computation is the computation of an object as defined by its class definition. For example, if an object is designed to interact with a device, the operations performed to meet that functionality define the base-level computation. The computation required to perform this computation is meta-level computation.

Meta-level Computation: Meta-level computation is the computation performed by an object's metaspace.

Reflection: The operation of an object viewing or communicating with its meta-system, or environment that it executes in, is called reflection [4, 24].

Reflective Computing: Reflective computing allows an object to query and alter its meta-functions, usually represented as metaobjects, during its execution and in a dynamic manner. An object and its metaobjects are causally connected, and the internals of the object are represented as metaobjects [6].

Reflector: A reflector is an object which provides the means and paths for communication between an active object and its metaspace.

Implicit Reflection: Implicit reflection is the result of an action of an object causing some meta-computation on its behalf [4]. Examples of this would be inter-object communication, or virtual memory paging facilities.

Explicit Reflection: Explicit reflection is the action of an object sending a message to one of its metaobjects [4]. An example of this in Apertos is an active object requesting to its metaspace for the identifier of another active object.

Declarative Reflection: Declarative reflection is the action of an object explicitly defining the behavioral constraints of its metaspace, on how it should behave, and how the metaspace should support it. An example of declarative reflection is an object declaring that it needs real-time support [39].

Meta-hierarchy: A meta-hierarchy is defined by the relationship between an object and a metaobject. Metaobjects are defined as objects, which also require a metaspace. In theory the meta-hierarchy can expand infinitely, though in Apertos, this is limited to a hierarchy of objects, metaobjects, and meta-metaobjects.

Message Passing: can mean two things in the object paradigm. Where passive objects are used, this can be as simple as the object-oriented equivalent to a function call in structured languages. This is what is done between standard

C++ objects. For active objects, this is analogous to message passing in traditional operating systems, requiring context switches, copying data between address spaces, etc., only with more support from the programming language to provide (e.g.) type checking.

A.2 Definitions Relating to Real-time Criteria

Response Time: The response time of a software system is the time between the presentation of a set of inputs to the software system and the appearance of all the associated outputs [45].

Real-time System: A real-time system is a system that must satisfy explicit (bounded) response-time constraints [40, 45]. The correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced.

Start Time: The start time is optional in defining the real-time constraints of a task. It specifies the earliest possible time a task can begin executing [45].

Execution Time: The execution time, or *computation time* specifies the maximum time required for a task to complete its execution [41].

Deadline: A deadline specifies when a task must complete [41].

Laxity: Laxity is deadline minus the sum of start time and execution time [45]. The laxity of a task is a measure of its urgency [41]. A task with zero laxity must be executed immediately and without interruption. A negative laxity indicates that a deadline will be missed.

Earliest Deadline First: Earliest deadline first is an optimal scheduling algorithm for single processor systems. Tasks are scheduled in order of deadlines [41].

Least Laxity: Least laxity is another optimal scheduling algorithm for uniprocessor

systems. Tasks are scheduled based on their remaining laxity [41].

A.3 Definitions Relating to Distributed Systems

Distributed System: A distributed system consists of multiple autonomous processors that do not share primary memory, but cooperate by messages over communication networks [1].

RPC, or Remote Procedure Call: RPC is a mechanism which allows programs to call procedures located on other machines. When a process on machine *A* calls a procedure on machine *B*, the calling process is suspended, and the execution of the called procedure takes place on *B*. Information can be transported from the caller to the callee in the parameters, and can come back in the procedure result. No message passing or I/O is at all visible to the programmer [23, 14].

Appendix B

Steps in Communications between Active Objects in Apertos

This appendix overviews all the steps performed for two application objects to communicate in version 0.5.1 of Apertos. Figure B.1 is a graphical representation of these steps.

B.1 Execution Associated with a Requesting Object

This section discusses the steps performed by the requesting object itself, when it begins or completes communication with another active object. An object may wish to send either an asynchronous or synchronous message to another object, or return a value from a synchronous call to the caller object, which requires communication with its reflector. All communication from an object to its reflector is done using the MetaCore primitive M. For this example, communication with the reflector MMiniMeta is assumed for simplicity. If communication were with either MZero or MCOOP, other objects would need to be used, although the basic operations would still be the same. Precise implementation details is available in the following source files:

```
$(TOP)/System/h/MMiniMetaMessage.h
$(TOP)/System/h/MetaCore.h
$(TOP)/System/lib/MMiniMeta/MMiniMetaFindMsg::FindByName.cc
$(TOP)/System/lib/MMiniMeta/MMiniMetaFindMsg::FindBySymbol.cc
$(TOP)/System/lib/MMiniMeta/MMiniMetaFindMsg::MMiniMetaFindMsg.cc
$(TOP)/System/lib/MMiniMeta/MMiniMetaFindMsg::MMiniMetaFindMsg1.cc
$(TOP)/System/lib/MMIniMeta/MMiniMetaFindMsg::MMiniMetaFindMsg2.cc
$(TOP)/System/lib/MMiniMeta/MMiniMetaMsg::Call.cc
$(TOP)/System/lib/MMiniMeta/MMiniMetaMsg::Send.cc
```

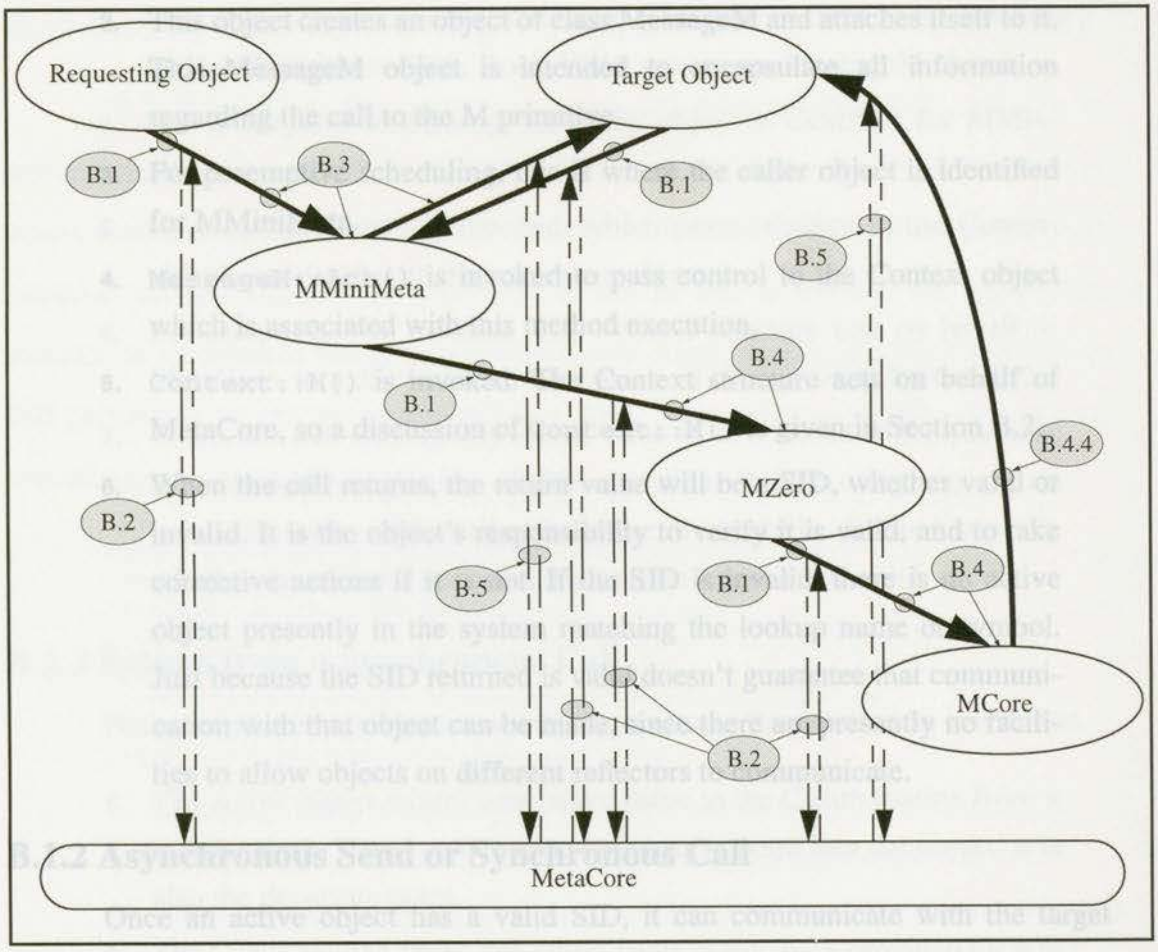


Figure B.1 Graphical Representation of Communications Discussed

```
$(TOP)/System/lib/MMiniMeta/MMiniMetaReplyMsg::Reply.cc
$(TOP)/System/lib/MetaCore/MessageM::Act.cc
$(TOP)/System/lib/MetaCore/MessageM::Initialize.cc
```

B.1.1 Finding the Target Object

Before an object can communicate with another active object, it must first find a handle by which to address the recipient object. This handle is called a SID, which stands for *S*hort *ID*entifier. For an object to ask the system for a valid SID, the following steps must be performed:

1. An object of class MMiniMetaFindMsg is created.

2. This object creates an object of class MessageM and attaches itself to it. This MessageM object is intended to encapsulate all information regarding the call to the M primitive.
3. For preemptive scheduling, this is where the caller object is identified for MMiniMeta.
4. `MessageM::Act()` is invoked to pass control to the Context object which is associated with this method execution.
5. `Context::M()` is invoked. The Context structure acts on behalf of MetaCore, so a discussion of `Context::M()` is given in Section B.2.
6. When the call returns, the return value will be a SID, whether valid or invalid. It is the object's responsibility to verify it is valid, and to take corrective actions if it is not. If the SID is invalid, there is no active object presently in the system matching the lookup name or symbol. Just because the SID returned is valid doesn't guarantee that communication with that object can be made, since there are presently no facilities to allow objects on different reflectors to communicate.

B.1.2 Asynchronous Send or Synchronous Call

Once an active object has a valid SID, it can communicate with the target object. This communication involves the following steps:

1. The requesting object constructs its message to be sent via MetaCore's M primitive. This message is comprised of parameter values for the target object, and room for any return value, if one is expected. No information with respect to the type of call, whether it be synchronous or asynchronous, is given.
2. An object of class MMiniMetaMsg is created. This object packages the structure which stores the parameters and room for the return value. Information with respect to if it's a synchronous or asynchronous call is given here.
3. The MMiniMetaMsg object creates an object from class MessageM and attaches itself to it. This MessageM object is intended to encapsulate all information regarding the call to the M primitive. The MMin-

B.2 Execution Resulting From a Call to Context::M()

At this point, an object has fully constructed its request to its reflector, to invoke MetaCore's M primitive. Control has been passed to the Context structure which acts on behalf of the MetaCore for this operation. Briefly, Context's task is to construct an environment the reflector can run in and to execute the appropriate method, as specified in the associated MessageM structure. This activity, however, is still performed on the requesting object's stack. Files relevant to this sequence of actions are:

```
$(TOP)/System/$(ARCH)/common/MetaCore/MetaCore_asm.s
$(TOP)/System/$(ARCH)/h/Context.h
$(TOP)/System/$(ARCH)/h/MetaCore.h
$(TOP)/System/$(ARCH)/lib/Context/Context::dispatchReadyContext.s
$(TOP)/System/$(ARCH)/lib/Context/Context::Activate.s
$(TOP)/System/$(ARCH)/lib/Context/Context::ExecuteM.cc
$(TOP)/System/$(ARCH)/lib/MetaCore/_Exec.s
$(TOP)/System/lib/Context/Context::M.cc
```

1. Execution starts in `Context::M()`. At this point, either information is to be sent or information is to be returned. This section applies to both.
2. Context information is saved, and `Context::ExecuteM()` is called.
3. Preparation to pass control to the target reflector is complete. This includes finding a stack upon which the reflector can execute, and determining whether a trap is required to perform a context switch.
4. If a trap is required (since the memory address space is to change), the message is packaged in a MessageCActivate structure, and the trap occurs. `_Exec()` performs the actual trap, to call `MetaContext::Activate()`.
5. If a trap is not required, `Context::Activate()` is executed. Both `Context::Activate()` and `MetaContext::Activate()` call `Context::executeReadyForM()`.
6. `Context::executeReadyForM()` is invoked, which calls `Context::dispatchReadyContext()`. This function performs the actual jump to the requested method in the reflector (in this case MMiniMeta).

B.3 Execution within Reflector MMiniMeta

The reflector's purpose is to provide a reliable, consistent link between an active object and its metaobjects, which comprise its metaspace. The basic steps used to start operations within MMiniMeta also apply to the starting or resuming of any active object; the major difference is in how the object returns. If the object is a reflector, then it follows pseudo-code similar to that presented here. If the object is an application object, see Section B.1.3.

Presently, there are three primitives within MMiniMeta that are of interest: `Call`, `Send`, and `Reply`. There is currently no provisions for applications objects to communicate across metaspaces, resulting in a greatly simplified operation. Related source files include:

```
$(TOP)/System/h/MMiniMeta.h
$(TOP)/System/h/MMiniMetaMessage.h
$(TOP)/System/common/MMiniMeta/MMiniMeta::Call.cc
$(TOP)/System/common/MMiniMeta/MMiniMeta::Find.cc
$(TOP)/System/common/MMiniMeta/MMiniMeta::Reply.cc
$(TOP)/System/common/MMiniMeta/MMiniMeta::Send.cc
$(TOP)/System/common/MMiniMeta/MMiniMeta_subr.cc
$(TOP)/System/common/MMiniMeta/_MMiniMeta_stub.cc
```

1. As with most objects, when control enters MMiniMeta, it enters via a C stub routine.
2. Each stub verifies that the message sent to it is compatible with the method being called. If not, an error status is set (stored in the MMiniMetaMsg object), and the actual call is skipped.
3. Once the call occurs, any error status returned by the call is stored in the MMiniMetaMsg object, and execution regarding MMiniMeta is halted with a call to `MZero::Exit()`. This operation occurs in much the same manner as that outlined in Section B.1.
4. If `MZero::Exit()` ever returns, there is either a problem or there is no other object is ready to execute; error/return codes are set appropriately when this occurs.

5. For preemptive scheduling, MMiniMeta's local variable, `activeObject`, is updated before the actual call is to `MMiniMeta::Call()`, `MMiniMeta::Send()`, and so forth, is made.
6. If control returns to the C stub routine, an object of type MessageR is created to invoke `Context::R()`. See Section B.5.

B.3.1 MMiniMeta::Call()

This method implements a synchronous call to another application object.

1. MMiniMeta places the caller object into a `WAIT` state, pending return of the synchronous call action.
2. The sender is specified so that MMiniMeta knows where the return value should be delivered.
3. If the target object is in a `DORMANT` state, it is placed in the `READY` state and added to MMiniMeta's private ready queue.
4. If the target object is in any other state (such as `READY`, `RUNNING`, or `WAIT`), the message is stored on the target object's message queue.
5. `MMiniMeta::reschedule()` is called. See B.3.5.
6. If control continues, there has been a failure, and control returns to the C stub routine to store the error code and call MZero.

B.3.2 MMiniMeta::Reply()

This method implements the actions needed to return a value from the target object to the original requesting object.

1. The return code for the returning target object has already been stored in the proper structure provided by the caller object; consequently, MMiniMeta doesn't have to address this issue.
2. If there are any pending messages for the returning target object, it is placed back into the `READY` state and onto MMiniMeta's ready queue. Otherwise, it is placed in the `DORMANT` state.
3. If a sender is specified, this reply is a result of a synchronous action, and the caller is placed back into the `READY` state and added to MMiniMeta's ready queue.

4. If a sender is not specified, this reply is a return from an asynchronous send, and no action regarding this is needed, other than to delete the associated memory allocated by `MMiniMeta::Send()`.
5. `MMiniMeta::reschedule()` is called. Refer to B.3.5.
6. If control returns, there has been a failure, and control returns to the C stub routine in order to store the error code and call MZero.

B.3.3 MMiniMeta::Send()

This method implements the asynchronous send provided by MMiniMeta.

1. The message to be sent to the target object is copied so the caller object can immediately resume execution without worrying about stack usage, retaining related structures, and so forth. No sender is specified since no return value is expected by the caller. (MCOOP supports returning values from asynchronous sends to the requesting object, MMiniMeta doesn't.)
2. If the target object is in a `DORMANT` state, it is placed in the `READY` state and added to MMiniMeta's ready queue.
3. If the target object is in any other state, the message is stored on the target object's message queue.
4. MMiniMeta returns to the C stub routine.

B.3.4 MMiniMeta::Find()

This method is invoked when an application object needs a SID to communicate another object.

1. While the lookup for another active object is being performed, the requesting object is placed in a `METAWAIT` state.
2. Unless an object is requesting its own SID value, MMiniMeta requests MZero to perform the name lookup. This request is done in a manner consistent with what is described in Section B.1.1. The value is returned using the R primitive, similar to how the M primitive is used. Refer to Section B.5 for details on implementing the R primitive.

B.3.5 MMiniMeta::reschedule()

This method is private to MMiniMeta. Its task is to add objects which are ready to execute to Exec's run queue. Unless a failure occurs, this method doesn't return.

B.3.5.1 Implementation for Nonpreemptive Scheduling

1. If there exists any objects on the ready queue, the first one is taken.
2. If there is any object in MMiniMeta's run queue, it is added to the root ready queue, managed by Exec. This request is done via MZero, in a manner consistent with that outlined in Section B.1.2. If an object's execution is to be resumed (from a reply) an entry point cannot be specified, so execution will resume where it suspended, as specified in `Context::M()`.
3. The local variable `activeObject` is updated to the object which was just added to Exec's ready queue.
4. `MZero::Exit()` is called in a manner consistent with that outlined in B.1.2.

B.3.5.2 Implementation for Preemptive Scheduling

1. For preemptive scheduling, any and all objects in MMiniMeta's ready queue are handled.
2. Each object is added to the root ready queue managed by Exec. This request is done through MZero, in a manner consistent with that in B.1.2.
3. `MZero::Exit()` is called in a manner consistent with that outlined in B.1.2.

B.4 Execution Within Reflectors MZero and MCore

The MZero reflector manages all other reflectors in Apertos, including MCore. MCore provides a metaspace for MZero, for consistency in the hierarchy. Here, the basic minimum to illustrate communication between two application objects is out-

lined. For more detail, refer to files:

```
$(TOP)/System/$(ARCH)/common/Exec/h/Activity.h
$(TOP)/System/$(ARCH)/common/Exec/h/Exec.h
$(TOP)/System/$(ARCH)/common/Exec/Activity.cc
$(TOP)/System/$(ARCH)/common/Exec/Exec.cc
$(TOP)/System/$(ARCH)/common/MZero/MZero_depend.cc
$(TOP)/System/common/MZero/h/ObjectMap.h
$(TOP)/System/common/MZero/MZero_public.cc
$(TOP)/System/common/MZero/MZero_subr.cc
$(TOP)/System/h/MetaCore.h
$(TOP)/System/h/MZero.h
$(TOP)/System/h/MZeroMessage.h
$(TOP)/System/lib/MetaCore/MessageR::Act.cc
$(TOP)/System/lib/MetaCore/MessageR::Initialize.cc
$(TOP)/System/lib/MZero/MZeroExitMsg::Exit.cc
$(TOP)/System/lib/MZero/MZeroFindMsg::Find.cc
```

B.4.1 MZero::Exit()

This method is only called by dependent application reflectors, and is used to signal MCore that another object can execute.

- MZero places the caller's state to `DORMANT` and calls `MZero::reschedule()`. If `MZero::reschedule()` returns, the caller is placed back into the `RUNNING` state, and returns an error code.

B.4.2 MZero::Find()

This method is called whenever any object in the system needs a SID to communicate with another object in the system.

1. MZero maintains a list of all objects which allow general communication with other objects, such as the Clock object introduced in this thesis. This list doesn't contain objects such as the virtual memory manager object, since these objects only provide access to a specific set of objects.
2. When a request is made, MZero looks up the object in its table. If it exists, the appropriate valid SID is returned, otherwise an invalid SID is returned.

B.4.3 MZero::reschedule()

This method is private to MZero, and provides similar functionality to

`MMiniMeta::reschedule()`.

1. If there exists any objects on MZero's ready queue, the first one is placed on Exec's ready queue.
2. `MCore::Exit()` is called in a manner consistent with B.1.2 (forever).

B.4.4 MCore::Exit()

This method is only method with can start the execution of any target object in version 0.5.1 of Apertos.

1. MCore queries Exec for the next object to execute by calling `Exec::Next()`. If no object can be executed, the Idle object is added to Exec's ready queue and this step is repeated. It is very seldom when only the Idle object can execute.
2. When an object which can be executed is found, that object's Activity is resolved, and the method `Activity::Run()` is called. This method constructs a MessageR structure, and a sequence of events similar to that of MessageM takes place, resulting in `Context::R()` being called. See B.5.
3. If the Idle object cannot be found, `MCore::Exit()` returns. If this happens, a major system failure has occurred.

B.5 Execution Resulting From a Call to Context::R()

At this point, a request to start or resume an object's execution has been made, via MetaCore's R primitive. Control has been passed to the Context structure which acts on behalf of MetaCore for this operation. Briefly, its task is to construct an environment the object can execute in and invoke the reflector's appropriate method, as specified by the associated MessageR structure. This action, however, is performed on the requesting object's stack. The requesting object may be either MCore, or an object on MCore, such as an exception handler. Hierarchical scheduling must allow other objects in the system to invoke this primitive. Refer to files:

```
$(TOP)/System/$(ARCH)/common/MetaCore/MetaCore_asm.s
$(TOP)/System/$(ARCH)/h/Context.h
$(TOP)/System/$(ARCH)/h/MetaCore.h
$(TOP)/System/$(ARCH)/lib/Context/Context::dispatchReadyContext.s
$(TOP)/System/$(ARCH)/lib/Context/Context::Activate.s
$(TOP)/System/$(ARCH)/lib/Context/Context::ExecuteR.cc
$(TOP)/System/$(ARCH)/lib/MetaCore/_Exec.s
$(TOP)/System/lib/Context/Context::R.cc
```

1. Execution starts in `Context::R()`. At this point, either information is to be sent, resulting in a new execution, or information is to be returned, resulting in the re-starting of a previously suspended execution.
2. Context information is saved, and `Context::ExecuteR()` is called.
3. Preparation to pass control to the metaspace is complete. This includes finding a stack upon which the target object can execute, and determining whether a trap is required to perform a context switch.
4. If the target object was in a suspended state or method is not specified, the previously saved stack pointer is used to resume execution. Otherwise, the start of the object's stack is used.
5. If a trap is required (since the memory address space is to change), the message is again packaged in a MessageCActivate structure, and the trap occurs. `_Exec()` performs the actual trap, to call `MetaContext::Activate()`.
6. If a trap is not required, `Context::Activate()` is executed. Both `Context::Activate()` and `MetaContext::Activate()` call `Context::executeReadyForR()`.
7. `Context::executeReadyForR()` finally invokes the target object, either by invoking the appropriate C stub routine, or by resuming a suspended execution.

Appendix C

Adding a New Active Object to the System

This appendix discusses the steps involved in introducing new active objects to the system. Section C.1 covers what is needed to add an active object which is to be supported by MCore. Section C.2 overviews the addition of application objects to the system.

C.1 Core Active Objects

The MCore reflector is reserved for system objects including MZero. All objects directly supported by MCore, with the exception of MZero, share the same memory address space to increase performance and reduce communication overheads. Objects which are associated directly with MetaCore are also supported by MCore.

Some system objects, however, are not restricted to being placed on MCore, and device drivers seldom are. Exception handlers are included in this group since they belong on MDrive. Since MDrive has been removed from the current R3000 implementation of Apertos, exception handlers are presently placed on MCore.

C.1.1 Object Code

The programming language of choice for all system objects is currently C++. An active object can be designed like any C++ passive object, and can include the sharing of global data with other Core objects (since everything executes in the same memory address space). Sections D.1.1 and D.1.2 present example code which implement the real-

time clock object introduced to Apertos.

MC++ [35] may be used for initial development, but is not viable for all code development. The source code produced by MC++ would have to be converted to use the correct communication primitives, since the language does not support MCore as a reflector.

C.1.2 Object Interface

An entry table similar to that used by dynamic link libraries in traditional operating systems has to be constructed manually. Table entries point to C stub code which perform very basic verification and unpack parameters for use by the C++ method. Communicating objects not directly supported by MCore have to use this table. Communication between objects on MCore can be done by either using this table with MCore's message passing primitives, or by directly calling the object as if it were a passive object (for better performance).

When defining the interface to an active object, public methods have to be numbered. When a communicating object uses the method table for access to the active object, it specifies the number of the entry in the table to execute. MetaCore uses the information in this table to determine the entry point into the object. Table entries point to the C stub code, which then invoke the appropriate method in the target active object.

Passive objects must be defined for use by requesting active objects. These objects package parameters and encapsulate the system calls needed to request the invocation of the target object's method via the method table. The source code presented in Sections D.1.3 and D.1.4 package parameters and encapsulate system calls for objects making requests to the real-time clock.

C.1.3 Introducing the Object

For clarity, the new object should be placed in some reasonable location, either by creating a new directory for it or by adding it to an existing directory. For the clock object introduced, a new directory called `$(TOP)/System/common/Clock` was created. The Interrupt object, which invokes the Clock object, was placed in the existing directory `$(TOP)/System/$(ARCH)/common/Exception`.

For objects to be tied directly to MCore, the following files have to be modified:

```
$(TOP)/System/$(ARCH)/common/MCore/h/MCoreExtern.h
$(TOP)/System/$(ARCH)/common/MCore/GlobalVariables.cc
$(TOP)/System/$(ARCH)/common/MCore/Makefile
$(TOP)/System/$(ARCH)/common/MCore/MCore::Prologue.cc
```

Modifying these files ties an object to MCore's compilation. Since the object is tied to MCore's compilation, MCore's makefile is modified rather than introducing another makefile into the object's directory. `MCore::Prologue.cc` is where MCore's initialization takes place, and where the object is created at run-time. The new object's storage, Context (including interrupt masks, etc.), Activity, execution stack, method entry table, and system identifier are also created here. Finally, if the entire system including application objects is to know about this object (for communication purposes), MCore notifies MZero of the new object's existence, and an entry identifying the new object is added to MZero's name table.

C.2 Non-Core Active Objects

Non-Core objects refer to objects which are not placed on the MCore reflector. This includes user reflectors (MMiniMeta, MCOOP), application objects, device driver objects, etc. MDrive and any objects it supports will belong in this list when MDrive is reintroduced to Apertos. Only user objects will be considered here, though all actions discussed can be easily applied to any object not placed on MCore.

C.2.1 Object Code

Two languages are available for programming a user object: C++ and MC++. MC++ is an extension to C++; its compiler simply produces C++ code. The purpose of MC++ is to automatically build the required stub code for the methods that other active objects can call. To ensure the generated code is correct, the default reflector for the new object (currently limited to MMiniMeta and MCOOP) has to be specified. As shown in Section D.2.1, the keyword *active* is used in the class definition to identify an active object.

Unfortunately, there is no private or protected access to externally available methods. All externally available methods are available for all objects in the system. From the set of public methods, the keyword *active* is used to identify such a group of methods. The object constructor is not added to the list of active methods to prevent other objects from directly invoking it. The constructor is left for the system to invoke during object creation. For more detail, see [35]. Sections D.2.1 and D.2.2 give an example of an active application object written in MC++.

Source code can be written like any normal passive C++ object, with the exception that only instance variables can be used. No access to global or class variables is supported because there is no guarantee that objects of the same class will reside in the same memory address space (or even the same physical machine), to share access to instance or global variables.

C.2.2 Object Interface

If MC++ is used, the object's interface will automatically be produced, and will be similar to that given in D.1.3 and D.1.4. If C++ is used, this code will have to be manually generated, as outlined in C.1.2. Since, during development, method interfaces can change, it is advisable to use MC++ to ensure all related objects remain valid with interface specifications.

C.2.3 Introducing the Object

Introducing an object which is not to execute directly on MCore requires additional steps. More detail is found in [46].

C.2.3.1 Location of Source

Newly defined application objects should be placed in the Objects subdirectory. For each such object, four directories have to be made. They should follow this pattern:

```
$(TOP)/Objects/Applications/{NAME}
$(TOP)/Objects/Applications/{NAME}/lib
$(TOP)/Objects/$(ARCH)/Applications/{NAME}
$(TOP)/Objects/$(ARCH)/Applications/{NAME}/lib
```

Source code written by the programmer is placed in the first directory. The source codes produced by MC++ are placed in the subsequent directories. The library directories store definitions for passive objects used to encapsulate parameter lists and system calls for use by other objects calling the new object. Makefiles are placed in the last two directories.

C.2.3.2 Compilation Environment

The compilation environment has to be notified of the new object, its location in the directory hierarchy, and the number of entry points (active methods). Depending on the type of object, one of the three files listed here needs to identify the location of the object.

```
$(TOP)/conf/config/files
$(TOP)/conf/config/files.$(ARCH)
$(TOP)/conf/config/files.$(ARCH).$(MODEL)
```

For example, if the object is a generic application object, is should be placed in `$(TOP)/conf/config/files`. If it is a hardware-specific device driver, is should likely be placed in `$(TOP)/conf/config/files/$(ARCH)/$(MODEL)`.

Next, the object has to be added to the Apertos executable image itself. To do

this, the Apertos configuration file being used has to be modified to show the addition of the new object. Here, the reflector that the object is to executes on, its stack size, execution mode (user vs. supervisor), CPU priority, scheduling priority, and the number of selectors (entry points) are specified. The script `configure`, located in configuration directory, can then be run to produce a new makefile, which will produce the new image.

C.2.3.3 Makefiles

Two makefiles are required for each Apertos user object: one for the active object, and one for the library code, which other objects use to communicate to it. If the object was written to produce output, it likely inherits from the object ULike, otherwise from Object. The makefiles from the superclass should be used as a template for the new object's makefiles. If the object has no superclass, the makefiles from Object should be used as a template.

C.2.3.4 Shell Object

The above steps should add a new active object to a new Apertos image. However, the Shell object cannot yet communicate with the new object, since Shell has not yet been notified of it. To notify Shell, the file `$(TOP)/Objects/$(ARCH)/Applications/ClassInfo/Spec.mc` has to be modified.

Appendix D

Sample Code

D.1 A Real-time Clock

D.1.1 Clock.h

```
//      Clock.h
//
//      By Robert W. Bryce
//      Version 1.03
//      Started:          March 6, 1995
//      Last Modified:    March 28, 1995

#ifndef OBJECT_CLOCK_H
#define OBJECT_CLOCK_H

#include <Alarm.h>        // $(TOP)/Objects/h/Alarm.h
#include <LinkedList.h>   // $(TOP)/Objects/h/LinkedList.h
#include <MetaLib.h>      // $(TOP)/Objects/h/MetaLib.h
#include <SID.h>          // $(TOP)/System/h/SID.h


typedef longword          Handle;


class Clock {
    private:
        LinkedList        armedAlarmList, disarmedAlarmList;
        Julian            currentTime;

        long              intrcnt, alarmcnt;
        Boolean           b;              // these used by InterruptHandler();

        long              interruptcount; // how often to increment clock
        long              clockcount;     // how much to increment clock by
        long              alarmcount;     // how often to check alarms
                                /* these are only used by InterruptHandler(),
                                   and one of interruptcount and clockcount
                                   should be 1 */

        LinkedList *findAlarm (class AlarmLink *a);
        void insertArmedAlarm (class AlarmLink *a);

    protected:
        void reset (long granularity);

    public:
        Clock ();

                /* Maintenance Functions */
        Boolean Reset (long granularity = -1);
                                        // resets hardware and logical clock
```

```
                                        // if granularity==-1, no change made
        Boolean ResetTime (const Time time, long granularity = -1);
                                        // calls Reset() and resets real time
        void InterruptHandler ();       // interrupt handler to increment clock
        void CheckAlarms ();            // routine to check alarms; called by
                                        // interrupt handler at intervals
                                        // specified by granulatiry

                /* Public Functions for General Use */
        Time GetCurrentTime ();         // return current real time
        size_t NumArmedAlarms ();       // return # armed Alarms
        size_t NumAlarms ();            // return total # Alarms

        Alarm BindAlarm (const AlarmInfo &time, const SID sid, longword selector, Handle handle);
         Boolean AdjustAlarm (Alarm anAlarm, const AlarmInfo &time, const SID sid, longword selec-
tor, Handle handle);
        Boolean UnBindAlarm (Alarm anAlarm);
                                        // disposes of an alarm.  For
                                        // consistency, an alarm is not
                                        // automatically disposed of when
                                        // it goes off, even if it isn't
                                        // periodic
        AlarmStatus GetAlarmInfo (Alarm anAlarm);
};

#endif
```

D.1.2 Clock.cc

```
//      Clock.cc
//
//      By Robert W. Bryce
//      Version 1.03
//      Started:        March 6, 1995
//      Last Modified:  March 28, 1995


#include <Clock.h>              // h/Clock.h
#include <Link.h>               // $(TOP)/Objects/h/Link.h
#include <FastTimer.h>          // $(TOP)/Objects/h/FastTimer.h
#include <MetaLib.h>            // $(TOP)/Objects/h/MetaLib.h
#include <MCoreMsg.h>           // $(TOP)/System/$(ARCH)/h/MCoreMsg.h
#include <_MCore_stub.h>        // $(TOP)/System/$(ARCH)/common/MCore/h/_MCore_stub.h


Clock::Clock () {
        ResetTimer();

        Time ct;

        intrcnt                 = 0;
        alarmcnt                = 0;
        b                       = 0;

        clockvalue *cv          = ReadClock ();
        ct.milliSeconds         = 0;
        ct.second               = cv->second;
        ct.minute               = cv->minute;
        ct.hour                 = cv->hour;
        ct.day                  = cv->date;
        ct.month                = cv->month;
        ct.year                 = cv->year + 1900;
        ct.Normalize();

        currentTime             = ct;

        interruptcount = ClockGranularity() / currentTime.Granularity();
        if (!interruptcount) {          // Time structure is finer
              interruptcount = 1;       // grained than interrupt
              clockcount = currentTime.Granularity() / ClockGranularity();
        } else  clockcount = 1;

        reset(currentTime.Granularity());
}
```

```
void Clock::reset (long granularity) {
        if (granularity != -1) {
                alarmcount = ClockGranularity() / granularity;
                if (!alarmcount)        // we can't quite meet requested
                        alarmcount = 1; // granularity for checking alarms
        }
}

void Clock::InterruptHandler () {
        alarmcnt++;
        if (alarmcnt == alarmcount)
                alarmcnt = 0;

        if (!alarmcnt)          // remember we have to check alarms since
                b = 1;          // alarmcount interrupts have occurred since
                                // we last checked them.
        intrcnt++;
        if (intrcnt == interruptcount)
                intrcnt = 0;
                                // we count off interrupts no matter what
        if (intrcnt)    return; // if we don't increment the clock this intr.
        currentTime += clockcount;
                                // increment clock appropriately
                /* no point checking alarms more often than we increment
                   the clock! */
        if (b) {                // if we should check alarms...
                b = 0;
                CheckAlarms();
        }
}


Boolean Clock::Reset (long granularity) {
        reset(granularity);
        return 1;
}


Boolean Clock::ResetTime (const Time time, long granularity) {
        reset(granularity);
        currentTime = time;
        currentTime.Normalize();
        clockvalue cv;
        Time ct         = currentTime;
        cv.second       = ct.second;
        cv.minute       = ct.minute;
        cv.hour         = ct.hour;
        cv.day          = ct.workDay - 1;
        cv.date         = ct.day;
        cv.month        = ct.month;
        int i           = ct.year - 1900;
        cv.year         = i;
        SetClock(&cv);
        return 1;
}


Time Clock::GetCurrentTime () {
        Time t = currentTime;
        t.Normalize();
        return t;
}


size_t Clock::NumArmedAlarms () {
        return armedAlarmList.Size();
}


size_t Clock::NumAlarms () {
        return armedAlarmList.Size() + disarmedAlarmList.Size();
}

class AlarmLink : public Link {
```

```
    public:
        Julian          SignalTime;             // thing to sort timers on
        Boolean         StartTimeValid;
        Julian          StartTime;
        Boolean         IntervalTimeValid;
        Julian          IntervalTime;
        Boolean         Periodic;
        SID             sid;
        longword        selector;
        Handle          handle;
};

void Clock::CheckAlarms () {
        AlarmLink *p = (AlarmLink *)armedAlarmList.First();

        while (p) {
                if (p->SignalTime <= currentTime) {
                        p = (AlarmLink *)armedAlarmList.RemoveFirst();
                                                // time to signal the alarm
                        if (p->Periodic) {
                                p->SignalTime += p->IntervalTime;
                                insertArmedAlarm(p);
                        }
                        else    disarmedAlarmList.AddLast(p);
                                                // we don't discard any alarm after
                                                // it goes off in case user program
                                                // wants to reuse it
                        // signal alarm
                        static Handle h = p->handle;
                        MCoreSendMsg msmsg(p->sid, p->selector, &h);
                        MessageM msgM(MCORE_SEND, &msmsg, FALSE);
                        (void) msgM.Act();      // this seems to be broken

                }
                else return;
                p = (AlarmLink *)armedAlarmList.First();
        }
}

LinkedList *Clock::findAlarm (AlarmLink *a) {
        if (a) {
                if (armedAlarmList.Find(a))
                        return &armedAlarmList;
                if (disarmedAlarmList.Find(a))
                        return &disarmedAlarmList;
        }
        return NULL;
}

void Clock::insertArmedAlarm (AlarmLink *a) {
                // we sort on AlarmLink.SignalTime, since that is the next
                // time the alarm should go off.  If it is periodic,
                // this always refers to the next time an alarm will
                // go off, otherwise it is left untouched after the
                // alarm goes off it's one time.
        AlarmLink *p = (AlarmLink *)armedAlarmList.First();
        if (p) {
                AlarmLink *last = (AlarmLink *)armedAlarmList.Last()->Next();
                while (p->SignalTime <= a->SignalTime) {
                        p = (AlarmLink *)p->Next();
                        if (p == last)
                                break;
                }
                // manually add node here right in front of p
                a->Next(p);
                a->Prev(p->Prev());
                p->Prev()->Next(a);
                p->Prev(a);
        } else  armedAlarmList.AddFirst(a);
}

Alarm Clock::BindAlarm (const AlarmInfo &time, const SID sid, longword selector, Handle handle) {
        if ((!time.StartTimeValid) && (!time.IntervalTimeValid))
```

```
            return NULL;
        Julian j;
        if ((time.IntervalTimeValid) && (j > time.IntervalTime))
                return NULL;
        if ((!time.IntervalTimeValid) && (time.Periodic))
                return NULL;
        AlarmLink *a = new AlarmLink;
        if (a) {
                a->StartTimeValid       = 1;
                if (time.StartTimeValid)
                        a->StartTime    = time.StartTime;
                else    a->StartTime    = currentTime + time.IntervalTime;
                a->IntervalTimeValid    = time.IntervalTimeValid;
                a->IntervalTime         = time.IntervalTime;
                a->Periodic             = time.Periodic;
                a->SignalTime           = a->StartTime;
                a->sid                  = sid;
                a->selector             = selector;
                a->handle               = handle;
                insertArmedAlarm(a);
        }
        return (Alarm)a;
}

Boolean Clock::AdjustAlarm (Alarm anAlarm, const AlarmInfo &time, const SID sid, longword selec-
tor, Handle handle) {
        if ((!time.StartTimeValid) && (!time.IntervalTimeValid))
                return 0;
        Julian j;
        if ((time.IntervalTimeValid) && (j > time.IntervalTime))
                return 0;
        if ((!time.IntervalTimeValid) && (time.Periodic))
                return 0;
        AlarmLink *a = (AlarmLink *)anAlarm;
        LinkedList *ll = findAlarm(a);
        if (ll) {
                ll->RemoveLast(a);
                a->StartTimeValid       = 1;
                if (time.StartTimeValid)
                        a->StartTime    = time.StartTime;
                else    a->StartTime    = currentTime + time.IntervalTime;
                a->IntervalTimeValid    = time.IntervalTimeValid;
                a->IntervalTime         = time.IntervalTime;
                a->Periodic             = time.Periodic;
                a->SignalTime           = a->StartTime;
                a->sid                  = sid;
                a->selector             = selector;
                a->handle               = handle;

                insertArmedAlarm(a);
                return 1;
        }
        return 0;
}

Boolean Clock::UnBindAlarm (Alarm anAlarm) {
        AlarmLink *a = (AlarmLink *)anAlarm;
        LinkedList *ll = findAlarm(a);
        if (ll) {
                ll->RemoveLast(a);
                delete a;
                return 1;
        }
        return 0;
}

AlarmStatus Clock::GetAlarmInfo (Alarm anAlarm) {
        AlarmStatus as;
        AlarmLink *a = (AlarmLink *)anAlarm;
        LinkedList *ll = findAlarm(a);
        if (ll) {
                as.Valid                = 1;
                if (ll == &armedAlarmList)
                        as.Armed        = 1;
```

```
        else    as.Armed                = 0;
        as.Info.StartTimeValid          = a->StartTimeValid;
        as.Info.StartTime               = a->StartTime;
        as.Info.IntervalTimeValid       = a->IntervalTimeValid;
        as.Info.IntervalTime            = a->IntervalTime;
        as.Info.Periodic                = a->Periodic;
        as.SignalTime                   = a->SignalTime;
    }
    return as;
}
```

D.1.3 ClockMsg.h

```
/*
 * The Apertos Operating System
 *
 * Any code and mechanism in this module may not be used
 * in any form without permissions.
 * Copyright (C) 1994 SONY COMPUTER SCIENCE LABORATORY Inc.,
 * All Rights Reserved.
 */
/*
 * REVISION
 * $Date: 1994/02/08 07:31:21 $
 * $Revision: 1.2 $
 */
/*
 * generated by MC++
 */

/* Message Header (Clock) */

#ifndef _ClockMsg_h_DEFINED
#define _ClockMsg_h_DEFINED

#include <Alarm.h>      // $(TOP)/Objects/h/Alarm.h

/* Defining Method Id */

/* Clock Object Interface */
#define CLOCK_RESET     0
#define CLOCK_RESETTIME 1
#define CLOCK_CHECKALARMS       2
#define CLOCK_GETCURRENTTIME    3
#define CLOCK_NUMARMEDALARMS    4
#define CLOCK_NUMALARMS 5
#define CLOCK_BINDALARM 6
#define CLOCK_ADJUSTALARM       7
#define CLOCK_UNBINDALARM       8
#define CLOCK_GETALARMINFO      9
/* Number of public method Object Interface */
#define CLOCK_NSELECTOR 10


/* Defining Message Structures */


/* Clock::Reset */
struct ClockResetMsg
{
        /* member */
        Boolean         _mc_0; /* return value */
        long    _mc_1;

        /* constructor */
        inline  ClockResetMsg ();
        /* public interface */
        Boolean         Reset (SID, long granularity = -1);
};

/* constructor */
inline  ClockResetMsg::ClockResetMsg () ()


/* Clock::ResetTime */
```

```
struct ClockResetTimeMsg
{
        /* member */
        Boolean         _mc_0; /* return value */
        const Time      _mc_1;
        long    _mc_2;

        /* constructor */
        inline  ClockResetTimeMsg ();
        /* public interface */
        Boolean         ResetTime (SID, const Time time, long granularity = -1);
};

/* constructor */
inline  ClockResetTimeMsg::ClockResetTimeMsg () {}


/* Clock::InterruptHandler */
struct ClockInterruptHandlerMsg
{
        /* member */
        /* return type is void */

        /* constructor */
        inline  ClockInterruptHandlerMsg ();
        /* public interface */
        void    InterruptHandler (SID);
};

/* constructor */
inline  ClockInterruptHandlerMsg::ClockInterruptHandlerMsg () {}


/* Clock::CheckAlarms */
struct ClockCheckAlarmsMsg
{
        /* member */
        /* return type is void */

        /* constructor */
        inline  ClockCheckAlarmsMsg ();
        /* public interface */
        void    CheckAlarms (SID);
};

/* constructor */
inline  ClockCheckAlarmsMsg::ClockCheckAlarmsMsg () {}


/* Clock::GetCurrentTime */
struct ClockGetCurrentTimeMsg
{
        /* member */
        Time    _mc_0; /* return value */

        /* constructor */
        inline  ClockGetCurrentTimeMsg ();
        /* public interface */
        Time    GetCurrentTime (SID);
};

/* constructor */
inline  ClockGetCurrentTimeMsg::ClockGetCurrentTimeMsg () {}


/* Clock::NumArmedAlarms */
struct ClockNumArmedAlarmsMsg
{
        /* member */
        size_t  _mc_0; /* return value */

        /* constructor */
        inline  ClockNumArmedAlarmsMsg ();
        /* public interface */
        size_t  NumArmedAlarms (SID);
};
```

```
/* constructor */
inline  ClockNumArmedAlarmsMsg::ClockNumArmedAlarmsMsg () {}

/* Clock::NumAlarms */
struct ClockNumAlarmsMsg
{
        /* member */
        size_t  _mc_0; /* return value */

        /* constructor */
        inline  ClockNumAlarmsMsg ();
        /* public interface */
        size_t  NumAlarms (SID);
};

/* constructor */
inline  ClockNumAlarmsMsg::ClockNumAlarmsMsg () {}

/* Clock::BindAlarm */
struct ClockBindAlarmMsg
{
        /* member */
        Alarm   _mc_0; /* return value */
        /* arg _mc_1 too complex */
        const AlarmInfo         _mc_1;
        SID                     _mc_2;
        longword                _mc_3;
        Handle                  _mc_4;

        /* constructor */
        inline  ClockBindAlarmMsg ();
        /* public interface */
         Alarm    BindAlarm (SID, const AlarmInfo &time, const SID sid, longword selector, Handle
handle);
};

/* constructor */
inline  ClockBindAlarmMsg::ClockBindAlarmMsg () {}

/* Clock::AdjustAlarm */
struct ClockAdjustAlarmMsg
{
        /* member */
        Boolean         _mc_0; /* return value */
        Alarm   _mc_1;
        /* arg _mc_2 too complex */
        const AlarmInfo         _mc_2;
        SID                     _mc_3;
        longword                _mc_4;
        Handle                  _mc_5;

        /* constructor */
        inline  ClockAdjustAlarmMsg ();
        /* public interface */
         Boolean          AdjustAlarm (SID, Alarm anAlarm, const AlarmInfo &time, const SID sid,
longword selector, Handle handle);
};

/* constructor */
inline  ClockAdjustAlarmMsg::ClockAdjustAlarmMsg () {}

/* Clock::UnBindAlarm */
struct ClockUnBindAlarmMsg
{
        /* member */
        Boolean         _mc_0; /* return value */
        Alarm   _mc_1;

        /* constructor */
        inline  ClockUnBindAlarmMsg ();
        /* public interface */
        Boolean         UnBindAlarm (SID, Alarm anAlarm);
};
```

```
/* constructor */
inline  ClockUnBindAlarmMsg::ClockUnBindAlarmMsg () ()


/* Clock::GetAlarmInfo */
struct ClockGetAlarmInfoMsg
{
        /* member */
        AlarmStatus     _mc_0; /* return value */
        Alarm   _mc_1;

        /* constructor */
        inline  ClockGetAlarmInfoMsg ();
        /* public interface */
        AlarmStatus     GetAlarmInfo (SID, Alarm anAlarm);
};

/* constructor */
inline  ClockGetAlarmInfoMsg::ClockGetAlarmInfoMsg () ()



#endif  /* _ClockMsg_h_DEFINED */
```

D.1.4 ClockMsg.mc

```
/*
 * The Apertos Operating System
 *
 * Any code and mechanism in this module may not be used
 * in any form without permissions.
 * Copyright (C) 1994 SONY COMPUTER SCIENCE LABORATORY Inc.,
 * All Rights Reserved.
 */
/*
 * REVISION
 * $Date: 1994/02/08 07:31:25 $
 * $Revision: 1.2 $
 */
/*
 * generated by MC++
 */
#define META_CALL

/* Message Body (ClockAdjustAlarm) */

#include        <MetaError.h>   // $(TOP)/System/h/MetaError.h
#include        <Clock.h>       // $(TOP)/Objects/h/Clock.h
#include        <ClockMsg.h>    // $(TOP)/Objects/h/ClockMsg.h
#include        <MCOOPMessage.h>// $(TOP)/System/h/MCOOPMessage.h
#include        <MetaLib.h>     // $(TOP)/Objects/h/MetaLib.h

extern "C++"
Boolean                 ClockAdjustAlarmMsg::AdjustAlarm (SID _mc_rec, Alarm anAlarm, const AlarmInfo
&time, const SID sid, longword selector, Handle handle)
{
        _mc_1 = anAlarm;
        (void) ByteCopy ((byte*) &time, (byte*) &_mc_2, sizeof (time));
        _mc_3 = sid;
        _mc_4 = selector;
        _mc_5 = handle;

        MCOOPMsg msg (_mc_rec, CLOCK_ADJUSTALARM, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        (void) ByteCopy ((byte*) &_mc_2, (byte*) &time, sizeof (_mc_2));
        return _mc_0;

}
```

```
extern "C++"
Alarm    ClockBindAlarmMsg::BindAlarm (SID _mc_rec, const AlarmInfo &time, const SID sid, longword
selector, Handle handle)
{
        (void) ByteCopy ((byte*) &time, (byte*) &_mc_1, sizeof (time));
        _mc_2 = sid;
        _mc_3 = selector;
        _mc_4 = handle;

        MCOOPMsg msg (_mc_rec, CLOCK_BINDALARM, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        (void) ByteCopy ((byte*) &_mc_1, (byte*) &time, sizeof (_mc_1));
        return _mc_0;

}

extern "C++"
void     ClockCheckAlarmsMsg::CheckAlarms (SID _mc_rec)
{

        MCOOPMsg msg (_mc_rec, CLOCK_CHECKALARMS, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return;

}

extern "C++"
AlarmStatus      ClockGetAlarmInfoMsg::GetAlarmInfo (SID _mc_rec, Alarm anAlarm)
{
        _mc_1 = anAlarm;

        MCOOPMsg msg (_mc_rec, CLOCK_GETALARMINFO, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}

extern "C++"
Time     ClockGetCurrentTimeMsg::GetCurrentTime (SID _mc_rec)
{

        MCOOPMsg msg (_mc_rec, CLOCK_GETCURRENTTIME, sizeof (*this), this);
#ifdef  META_CALL
        mError m = msg.Call ();
        Printf("error code for ClockGetCurrentTimeMsg::Call = %d\n", m);
        Flush();
#else   /* META_CALL */
        mError m = msg.Send ();
        Printf("error code for ClockGetCurrentTimeMsg::Send = %d\n", m);
        Flush();
        m = msg.Receive ();
        Printf("error code for ClockGetCurrentTimeMsg::Receive = %d\n", m);
        Flush();
#endif  /* META_CALL */
        return _mc_0;

}

extern "C++"
size_t   ClockNumAlarmsMsg::NumAlarms (SID _mc_rec)
{
```

```
        MCOOPMsg msg (_mc_rec, CLOCK_NUMALARMS, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}
extern "C++"
size_t  ClockNumArmedAlarmsMsg::NumArmedAlarms (SID _mc_rec)
{

        MCOOPMsg msg (_mc_rec, CLOCK_NUMARMEDALARMS, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}

extern "C++"
Boolean         ClockResetMsg::Reset (SID _mc_rec, long granularity)
{
        _mc_1 = granularity;

        MCOOPMsg msg (_mc_rec, CLOCK_RESET, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}

extern "C++"
Boolean         ClockResetTimeMsg::ResetTime (SID _mc_rec, const Time time, long granularity)
{
        _mc_1 = time;
        _mc_2 = granularity;

        MCOOPMsg msg (_mc_rec, CLOCK_RESETTIME, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}

extern "C++"
Boolean         ClockUnBindAlarmMsg::UnBindAlarm (SID _mc_rec, Alarm anAlarm)
{
        _mc_1 = anAlarm;

        MCOOPMsg msg (_mc_rec, CLOCK_UNBINDALARM, sizeof (*this), this);
#ifdef  META_CALL
        (void) msg.Call ();
#else   /* META_CALL */
        (void) msg.Send ();
        (void) msg.Receive ();
#endif  /* META_CALL */
        return _mc_0;

}
```

D.1.5 Alarm.h

```
//      Alarm.h
//
//      By Robert W. Bryce
//      Version 1.02
//      Started:        March 20, 1995
//      Last Modified:  March 29, 1995

#ifndef OBJECT_ALARM_H
#define OBJECT_ALARM_H

#include <Realtime.h>   // $(TOP)/Objects/h/Realtime.h


typedef longword Alarm;


class AlarmInfo {
    public:
        Boolean     StartTimeValid;
        Time        StartTime;          // start time of alarm (optional)
        Boolean     IntervalTimeValid;
        Julian      IntervalTime;       // relative time of alarm (optional)
        Boolean     Periodic;           // whether alarm is once-off or repeats

        void copy (const AlarmInfo &toCopy);

        AlarmInfo ();
        AlarmInfo (const AlarmInfo &toCopy);

        AlarmInfo &operator = (const AlarmInfo &toCopy);
};


/*      Alarms:
        Either the start time, interval time, or both must be specified.
        If the start time is not specified, Clock calculates it by
        current_time + interval_time.
        The alarm will go off at the calculated/specified start time.  If
        Periodic is TRUE, then the alarm will continue to go off
        at IntervalTime intervals after StartTime.
*/

class AlarmStatus {
    public:
        Boolean         Valid;          // if info that follows if valid
        Boolean         Armed;          // if alarm is set and ready to go at some time
        AlarmInfo       Info;           // absolute time of alarm to go off

        Time            SignalTime;     // next/last time timer will go off

        void copy (const AlarmStatus &toCopy);

        AlarmStatus ();
        AlarmStatus (const AlarmStatus &toCopy);

        Boolean IsValid ()              { return Valid; };
        Boolean IsArmed ()              { return Armed; };
        AlarmInfo &GetAlarmInfo ()      { return Info; };

        AlarmStatus &operator = (const AlarmStatus &toCopy);
};

#endif
```

D.1.6 Realtime.h

```
//      Realtime.h
//
//      By Robert W. Bryce
//      Version 1.01
//      Started:        March 8, 1995
//      Last Modified:  March 28, 1995
```

```
#ifndef OBJECT_REALTIME_H
#define OBJECT_REALTIME_H

#include <Types.h>       // $(TOP)/System/h/Types.h

#ifdef __mcplusplus
#include <OutStream.h>  // $(TOP)/Objects/h/OutStream.h
#endif

typedef enum {
        Sunday = 0, sunday = 0,
        Monday = 1, monday = 1,
        Tuesday = 2, tuesday = 2,
        Wednesday = 3, wednesday = 3,
        Thursday = 4, thursday = 4,
        Friday = 5, friday = 5,
        Saturday = 6, saturday = 6
) Weekday;


class Time;

class Julian {
    public:
        long    milliseconds;
        long    days;

        Julian (long ms = 0, long d = 0);
        Julian (const Julian &toCopy);
        Julian (const Time &toCopy);

        void Normalize ();

        Julian operator + (long ms) const;
        Julian operator + (const Julian &toAdd) const;
        Julian operator + (const Time &toAdd) const;
        Julian operator - (long ms) const;
        Julian operator - (const Julian &toSubtract) const;
        Julian operator - (const Time &toSubtract) const;
        Julian &operator += (long ms);
        Julian &operator += (const Julian &toAdd);
        Julian &operator += (const Time &toAdd);
        Julian &operator -= (long ms);
        Julian &operator -= (const Julian &toSubtract);
        Julian &operator -= (const Time &toSubtact);
        Julian &operator = (const Julian &toCopy);
        Julian &operator = (const Time &toCopy);
        Boolean operator == (const Julian &toCompare) const;
        Boolean operator == (const Time &toCompare) const;
        Boolean operator != (const Julian &toCompare) const;
        Boolean operator != (const Time &toCompare) const;
        Boolean operator < (const Julian &toCompare) const;
        Boolean operator < (const Time &toCompare) const;
        Boolean operator <= (const Julian &toCompare) const;
        Boolean operator <= (const Time &toCompare) const;
        Boolean operator > (const Julian &toCompare) const;
        Boolean operator > (const Time &toCompare) const;
        Boolean operator >= (const Julian &toCompare) const;
        Boolean operator >= (const Time &toCompare) const;

        static long Granularity()       { return 1000; };
                                        // accurate to 1/1000th second
};


class Time {
    friend class Julian;
    private:

#ifdef __mcplusplus
        void displayday(OutStream &s) const;
        void displaymonth(OutStream &s) const;
        void displayyear(OutStream &s) const;
#endif

    public:
```

```
        long            milliSeconds;
        short           second;
        short           minute;
        short           hour;           // this and above origin 0
        short           day;            // this and below origin 1
        short           month;
        short           year;
        short           yearDay;        // what day of the year (1...365)
        Weekday         workDay;        // what day of the week it is

    protected:
        short           currentFormat;

        void copy (const Time &toCopy);

        void doSeconds (long milliseconds);
        long normalizeDay ();

    public:
        Time (const Time &toCopy);
        Time (  long    ms = 0,
                short   s = 0,
                short   mi = 0,
                short   h = 0,
                short   d = 1,
                short   mo = 1,
                short   y = 1995,
                short   format = 0x0001);       // FORMAT_CANADIAN default (see below)
        Time (const Julian &date, short format = 0x0001);

        void Normalize ();

#ifdef __mcplusplus
        void Display (OutStream &s) const;
        void DisplayDate (OutStream &s) const;
        void DisplayTime (OutStream &s) const;

        void SetDisplayFormat (short format);
        short GetDisplayFormat ();
                // legal formats:
            #define FORMAT_CANADIAN             (0x0001)        // dd/mm/yy
            #define FORMAT_LONGCANADIAN         (0x0101)        // dd/mm/yyyy
            #define FORMAT_TEXTCANADIAN         (0x0201)        // dd mmm yy
            #define FORMAT_LONGTEXTCANADIAN     (0x0301)        // dd mmm yyyy
            #define FORMAT_USA                  (0x0002)        // mm/dd/yy
            #define FORMAT_LONGUSA              (0x0102)        // mm/dd/yyyy
            #define FORMAT_TEXTUSA              (0x0202)        // mmm dd yy
            #define FORMAT_LONGTEXTUSA          (0x0302)        // mmm dd yyyy
            #define FORMAT_YEAR                 (0x0004)        // yy/mm/dd
            #define FORMAT_LONGYEAR             (0x0104)        // yyyy/mm/dd
            #define FORMAT_TEXTYEAR             (0x0204)        // yy mmm dd
            #define FORMAT_LONGTEXTYEAR         (0x0304)        // yyyy mmm dd
            #define FORMAT_YEAR1                (0x0008)        // yy/dd/mm
            #define FORMAT_LONGYEAR1            (0x0108)        // yyyy/dd/mm
            #define FORMAT_TEXTYEAR1            (0x0208)        // yy dd mmm
            #define FORMAT_LONGTEXTYEAR1        (0x0308)        // yyyy dd mmm

            #define FORMAT_DATEFIRST            (0x1000)        // put date before time
            #define FORMAT_TIMEFIRST            (0x0000)        // put time before date
                                                                // (default)
            #define FORMAT_INCLUDEDAYOFWEEK     (0x2000)        // put day of week in
#endif

        Time operator + (long ms) const;
        Time operator + (const Julian &toAdd) const;
        Time operator + (const Time &toAdd) const;
        Time operator - (long ms) const;
        Time operator - (const Julian &toSubtract) const;
        Time operator - (const Time &toSubtract) const;
        Time &operator += (long ms);
        Time &operator += (const Julian &toAdd);
        Time &operator += (const Time &toAdd);
        Time &operator -= (long ms);
        Time &operator -= (const Julian &toSubtract);
        Time &operator -= (const Time &toSubtact);
        Time &operator = (const Julian &toCopy);
        Time &operator = (const Time &toCopy);
```

```
        Boolean operator == (const Julian &toCompare) const;
        Boolean operator == (const Time &toCompare) const;
        Boolean operator != (const Julian &toCompare) const;
        Boolean operator != (const Time &toCompare) const;
        Boolean operator < (const Julian &toCompare) const;
        Boolean operator < (const Time &toCompare) const;
        Boolean operator <= (const Julian &toCompare) const;
        Boolean operator <= (const Time &toCompare) const;
        Boolean operator > (const Julian &toCompare) const;
        Boolean operator > (const Time &toCompare) const;
        Boolean operator >= (const Julian &toCompare) const;
        Boolean operator >= (const Time &toCompare) const;

#ifdef __mcplusplus
        friend OutStream &operator << (OutStream &s, const Time &time)
                { time.Display(s); return s; };
        friend OutStream &operator << (OutStream &s, const Time *time)
                { time->Display(s); return s; };
#endif

        static long Granularity ()      { return 1000; };
};                                      // accurate to 1/1000th second

#endif
```

D.2 A User Object

D.2.1 ThesisTest.h

```
//      ThesisTest.h
//
//      By Robert W. Bryce
//      Version 1.0
//      Started:        March 7, 1995
//      Last Modified:  April 27, 1995


#ifndef OBJECT_THESISTEST_H
#define OBJECT_THESISTEST_H


#include <Clock.h>          // $(TOP)/Objects/Applications/Clock/h/Clock.h
#include <FastTimer.h>      // $(TOP)/Objects/Applications/Clock/h/FastTimer.h
#include <LogicalTimer.h>   // $(TOP)/Objects/Applications/Clock/h/LogicalTimer.h
#include <ULike.h>          // $(TOP)/Objects/Applications/Object/h/ULike.h
#include <SilentTest.h>     // $(TOP)/Objects/Applications/SilentTest/h/SilentTest.h
#include <LoudTest.h>       // $(TOP)/Objects/Applications/LoudTest/h/LoudTest.h


active class ThesisTest : public ULike {
        Clock           *clock;     // an active object on MCore
        SID             scheduler;  // another active object on MCore
        SilentTest      *st;        // an active object on MMiniMeta
        LoudTest        *lt;        // another active object on MMiniMeta

    public:
        ThesisTest ();
    active:
        void Start ();
        void Display ();
        void FastTimerTest ();
        void PingPongTest ();
        void TimerTest ();
};

#endif
```

D.2.2 ThesisTest.mc

```
//      ThesisTest.mc
//
//      By Robert W. Bryce
```

```
//      Version 1.0
//      Started:        March 7, 1995
//      Last Modified:  May 3, 1995

#include <MCOOPMessage.h>    //$(TOP)/System/h/MCOOPMessage.h
#include <MetaLib.h>         // $(TOP)/Objects/h/MetaLib.h
#include <OutStream.h>       // $(TOP)/Objects/Applications/OutStream/h/OutStream.h
#include <ThesisTest.h>      // $(TOP)/Objects/Applications/Thesis/h/ThesisTest.h
#include <ClockMsg.h>        // $(TOP)/Objects/h/Clock.h
#include <SilentTestMsg.h>   // $(TOP)/Objects/Applications/SilentTest/h/SilentTest.h
#include <LoudTestMsg.h>     // $(TOP)/Objects/Applications/LoudTest/h/LoudTest.h


ThesisTest::ThesisTest () {
}


void ThesisTest::Start () {
        ULike::Start();
        ResetTimer();
                                // looking for other objects in the system
        clock = meta->Find("clock");
        if (clock.IsValid())
                        cout << "Found the Clock.\n";
        else    cout << "Couldn't find a clock.\n";

        scheduler = meta->Find("scheduler");
        if (scheduler.IsValid())
                cout << "Found the Scheduler.\n";
        else    cout << "Couldn't find a scheduler.\n";

        st = meta->Find("silentTest");
        if (st.IsValid()) {
                cout << "Found SilentTest\n";
                st->Start();
        } else  cout << "Couldn't find SilentTest\n";

        lt = meta->Find("loudTest");
        if (lt.IsValid()) {
                cout << "Found LoudTest\n";
                lt->Start();
        } else  cout << "Couldn't find LoudTest\n";
}

void ThesisTest::Display () {
        cout << "I'm the ThesisTest object!\n";
        st->Display();
        lt->Display();
}

void ThesisTest::FastTimerTest () {
        FastTimer ft;
        longword lw = ft.GetTime();
        FastTimer f2;
        for (long i=0; i < 0x000fffff; i++);
        longword l2 = f2.GetTime();
        cout << "Time 1: " << lw << "Time 2: " << l2 << "\n";
        Flush();
}

void ThesisTest::PingPongTest () {
                                // cout's produce synchronous sends to OutStream
        cout << "PingPongTest not yet complete.\n";

        cout << "about to send first asynchronous send...\n";

        LoudTestTestMsg lMsg;
        MCOOPMsg msg2(lt, LOUDTEST_TEST, sizeof(lMsg), &lMsg);
        msg2.Send();

        cout << "about to send second asynchronous send...\n";

        SilentTestTestMsg sMsg;
```

```
        MCOOPMsg msg1(st, SILENTTEST_TEST, sizeof(sMsg), &sMsg);
        msg1.Send();

        cout << "done!\n";
}

void ThesisTest::TimerTest () {
        Time tt;
        tt = clock->GetCurrentTime();
        cout << tt << "\n";
        Time t3;
        t3.Normalize();
        cout << "day of the week for " << t3 << " is " << t3.workDay << "\n";
        cout << "day of the year for " << t3 << " is " << t3.yearDay << "\n";
        t3.day = 22;
        t3.month = 3;
        t3.SetDisplayFormat(FORMAT_LONGTEXTCANADIAN | FORMAT_DATEFIRST | FORMAT_INCLUDEDAYOFWEEK);
        t3.Normalize();
        cout << "day of the week for " << t3 << " is " << t3.workDay << "\n";
        cout << "day of the year for " << t3 << " is " << t3.yearDay << "\n";

        Time t1(999,59,59,23,31,12,1995);
        cout << t1 << "\n";
        t1.Normalize();
        cout << t1 << "\n";
        t1 += 1;
        cout << t1 << "\n";
        Time t2(-50);
        cout << t2 << "\n";
        t2.Normalize();
        cout << t2 << "\n";
        tt = clock->GetCurrentTime();
        cout << tt << "\n";
        cout << "Done!";
}
```

D.2.3 ULike.h

```
/*
 * The Apertos Operating System
 *
 * Any code and mechanism in this module may not be used
 * in any form without permissions.
 * Copyright (C) 1994 SONY COMPUTER SCIENCE LABORATORY Inc.,
 * All Rights Reserved.
 */
/*
 * REVISION
 * $Date: 1993/10/08 03:07:07 $
 * $Revision: 1.11 $
 */

#ifndef _ULike_h_DEFINED
#define _ULike_h_DEFINED

/*
 *      UNIX Like
 *              cin, cout, cerr
 */

#include        <Object.h>      // Objects/h

#ifdef  __mcplusplus
active class    InStream;
active class    OutStream;
#else
class   InStream;
class   OutStream;
#endif

#ifdef  __mcplusplus
//active class ULike : public Object, meta MCOOP {
active class ULike : public Object {
#else
class ULike : public Object {
```

```
#endif
protected:
        InStream*       cin;
        OutStream*      cout;
        OutStream*      cerr;

protected:
public:
                ULike ();
#ifdef  __mcplusplus
active:
#endif
        void    Start ();
        void    Display ();
        void    Main ();
};

#endif  /* ULike_h_DEFINED */
```

VITA

Surname: Bryce Given Names: Robert William

Place of Birth: Treherne, Manitoba, Canada
Date of Birth: December 8, 1970

Educational Institutions Attended:

University of Victoria	1993 to 1995
Brandon University	1988 to 1992

Degrees Awarded:

B. Sc. (Specialist)	Brandon University	1993

Honors and Awards:

University of Victoria	
President's Award	1994
President's Award	1993
Natural Sciences and Engineering Research Council	
Post-Graduate Scholarship A	1993 to 1995
B. C. Advanced Systems Institute	
G.R.A.P. Scholarship	1993
Brandon University	
Certificate of Merit in Computer Science	1989

Publications:

1. G. D. Richards and R. W. Bryce. "A Computer Algorithm for Simulating the Spread of Wildland Fire Perimeters for Heterogeneous Fuel and Meteorological Conditions." In International Journal of Wildland Fire, 5(2):73-79, 1995.

2. R. W. Bryce, K. Murata, G. C. Shoja, and E. G. Manning. "Porting and Enhancements of a Real-time Object-Oriented Operating System." In the Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. pp. 606-609, 1995.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: ENHANCING REAL-TIME PERFORMANCE OF AN OBJECT-ORIENTED OPERATING SYSTEM

Author [redacted]
ROBERT WILLIAM BRYCE

SEPTEMBER 8, 1995

(Date)