Chameleon, a Dynamically Extensible and Configurable Object-Oriented Operating System

by

Robert William Bryce
B.Sc., Brandon University, 1992
M.Sc., University of Victoria, 1995

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

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

Dr. E. G. Manning, (Department of Computer Science)

Dr. M. H. M. Cheng, (Department of Computer Science)

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

Dr. G. D. P. Dueck, External Examiner (Department of Mathematics and Computer Science, Brandon University)

Supervisor: Dr. G. C. Shoja

ABSTRACT

Currently, new algorithms are being incorporated into operating systems to deal with a host of new requirements from multimedia applications. These new algorithms deal with soft real-time scheduling, different memory models, and changes to buffer caching and network protocols. However, old design techniques such as structured programming, global variables and implied dependencies are impeding this development and proof of correctness. Many current operating system research groups are developing extensible systems, where new code can be placed into the system and even kernel layers. A primary difficulty in these efforts is how to avoid adversely affecting reliability and traditional measures of performance.

Techniques from the object orientation paradigm are being incorporated to better manage these issues because they have shown promise in improving modularity, information hiding, and reusability. In some cases, these techniques are even being used to build fresh operating systems from the ground up with the goal of easier extensibility and adaptability in the future. The Apertos operating system introduced and implemented many concepts originally alien to operating system research but exhibited unacceptable performance for multimedia applications.

This dissertation introduces Chameleon, a new object-oriented operating system that shares the same philosophical approach as Apertos, leveraging *meta* designs and concepts to deal with the diverse requirements of today's and future multimedia applications. However, Chameleon takes a new and original approach to design and implementation to achieve a high degree of adaptability and retain the performance of a micro-kernel.

In Chameleon, the object-oriented paradigm serves as the basis for newly introduced concepts such as *AbstractCPU*, *brokers*, and the *broker interface hierarchy*. Together, AbstractCPU, brokers, and related software engineering techniques such as *dynamic class binding* serve as a basis for all system management, communication, and for an event-driven model where new events can be defined and dynamically introduced to a running system.

The *meta* design clearly defines a hierarchy of "operating environments" that can be optimized for a particular type of application. As such, hierarchical resource management plays an important role in Chameleon. A minimal set of primitives that is appropriate for hierarchical memory management is defined atop a single address space memory model. Similarly,

hierarchical CPU scheduling is employed, as different applications will exhibit different scheduling requirements. Different schedulers may then co-exist on the same CPU. Communication in a hierarchically structured operating system is also detailed.

The implementation of the Chameleon structuring concept is presented and analyzed. Standard performance measures are used to compare Chameleon to related research and commercial operating systems. Costs of individual operations are also presented to outline the overheads and gains associated with the Chameleon model.

Examiners:

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

Dr. E. G. Manning, (Department of Computer Science)

Dr. M. H. M. Cheng, (Department of Computer Science)

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

Dr. G. D. P. Dueck, External Examiner (Department of Mathematics and Computer Science, Brandon University)

Table of Contents

List of Tables

List of Figures

Glossary of Terms

API	application programming interface
BVT	borrowed-virtual-time
COM	component object model
CORBA	common object request broker architecture
DSM	distributed shared memory
IDL	interface definition language
IPC	inter-process communication
LRU	least recently used
MIPS	million instructions per second
MOP	meta-object protocol
MPEG	moving pictures experts group
MRU	most recently used
OID	object identifier
PID	process identifier
RTTI	run-time type information
SAS	single address space
SASOS	single address space operating system
SFI	software fault isolation

Acknowledgements

I would like to take this opportunity to sincerely thank my supervisor, Dr. G. C. Shoja for his advice and enduring patience throughout this Ph.D. programme. His guidance, encouragement, and insightful observations have made a remarkable improvement on this work. He has been instrumental, not only in the completion of this work, but my decision to actually complete this work.

I would like to thank NSERC for helping fund this degree.

I owe a debt of gratitude to Robert Macdonald. As an employer, he was always understanding and willing to allow me to use company supplies and resources when necessary, and permit me to take the time off work to complete this degree. As a friend, he has acted as a sounding board and helped point out obvious sources of errors and omissions that occur when one is too focused on a single aspect of the work.

Most importantly, I express my deepest thanks to my parents. They have been supportive for every aspect, and every step in my academic endeavours. Without them, and their contributions, this degree certainly would not have been possible.

Chapter 1

Introduction

The performance of contemporary computer hardware has made it possible for a new category of applications, called *multimedia applications*, to exist for the average user. In general, multimedia applications have characteristics that are not shared by traditional applications such as databases: they are generally continuous real-time, they involve large transfers of data (e.g., 4 Mbps for MPEG-2 compressed video), and quality of service often takes precedence over correctness of data contained in a network packet. Thus, many key areas of an operating system are affected. New scheduling and messaging solutions are essential for the system to operate efficiently [1, 2]. Changes in buffer caching and network protocols [3] are also necessary.

A primary problem, however, is that even though the hardware is capable of providing for these tasks, commodity operating systems are generally ill prepared for the new requirements of these applications. As such, some authors and researchers have proposed that these multimedia applications should be serviced by a separate operating system, working in unison with processes running on a host commodity (e.g. UNIX) system [4]. However, this is not a viable solution for the average user, and weaknesses in current commodity operating system designs have become apparent. Current operating system design philosophies: structured approaches, shared entities, global variables, and implied dependencies have made it difficult to fully integrate new functionality into a legacy system while maintaining complete correctness [5]. As well, in the interests of performance, some current micro-kernel systems are no longer "micro", either with respect to their memory sizes or with respect to their complicated interfaces [6].

Many research groups have been developing various operating system structuring techniques to address these issues, including virtual machine structuring in Fluke [1], micro-kernel structuring in L4 [7], Chorus [8] and Mach [9], proxy structuring in SOS [6], meta-structuring in PM [10] and Apertos [5], and others [6, 11]. Other research, such as Spin [12] and Vino [13] have modified existing abstractions, such as micro-kernel, to improve extensibility [14]. The prime difficulty is to avoid adversely affecting reliability or traditional measures of performance, such as inter-process communication, while increasing operating system extensibility and flexibility to make it easier to introduce new functionality.

1.1 Object-Oriented Methodology

To address the design shortcomings of current operating systems, object-oriented methodologies are also slowly being incorporated. Object-oriented techniques have shown promise because of their powerful characteristics, including modularity, information hiding, and reusability, and their tendancy to produce more reliable and easier to maintain code than traditional programming paradigms. Some systems, such as Apertos [15] and PM [10], were designed so that the entire system was based on object-oriented methodologies with no compromises to the paradigm. However, Apertos, for example, was never practical for commercial use given the timing constraints and other requirements of today's multimedia applications. It suffered from inefficient communication, and also lacked basic features such as preemptive and real-time scheduling. In its simplest form, its communication performance was far behind its contemporaries [16] and would continue to degrade almost exponentially as the system grew due to its recursive structure involving many context switches to handle basic message sends.

1.2 Goals and Objectives

This dissertation introduces a new operating system structure called Chameleon[1]. Some of Chameleon's goals are similar to those of Apertos, including well-defined terminologies and inter-dependencies, support for diverse requirements from different applications and a pure object-oriented paradigm. Chameleon, however, has the additional goals, namely achieving performance that is comparable to contemporary micro-kernel operating systems, and dynamic reconfigurability to introduce new functionality and for easy revision control. The system is built up in response to user requirements to accommodate the particular applications being executed. Applications execute in specialized operating environments called *meta-spaces*, which support different run-time requirements. Although solutions are designed with the diverse requirements of multimedia applications in mind, efforts have been made to ensure that they remain suitable for supporting more traditional applications, such as databases.

These goals have been met in every respect. The focus of this work is therefore to provide and demonstrate a clean object-oriented operating system structuring concept which meets today's performance expectations. It must also attain the extensibility and flexibility necessary for a general-purpose operating system design to support multimedia requirements, such as quality of service and soft real-time scheduling, in a manner similar to purpose-built systems.

---

[1]The name 'Chameleon' was chosen because the operating system exhibits properties analogous to the animal: although it remains the same, it can adapt to its environment (albeit instead of changing colours, the operating system can change functionality, and provide different policies simultaneously).

1.3 Contributions and Unique Features of Chameleon

Chameleon's components have been carefully selected and designed to minimize redundancy in the system and to maintain a high degree of consistency in the architecture, while obtaining the goals listed above. The object-oriented paradigm is extended and exploited to provide consistency to all aspects of the Chameleon structuring concept. Terminologies such as *meta-space* that were taken from programming language research and applied to operating system research, are revisited, scrutinized, and clarified.

Chameleon introduces new concepts, called *AbstractCPU* and the *broker interface hierarchy*, where *brokers* are the principal tools for both general system management, and for specialized abstract environments tailored to attain optimal performance for different types of applications. AbstractCPU defines and provides an environment for all brokers to execute in. It also plays an important role in all communications among protected entities, and as such defines the communication semantics for the entire system. It serves as a basis for an event-driven model in which very few events are pre-defined. The rules and means by which AbstractCPU links to brokers is a new and unique concept in operating system design and is called *dynamic class binding.*

The broker interface hierarchy design effectively exploits the object model to attain consistency in design and a high degree of extensibility and reconfigurability. This design is very different from conventional kernel and micro-kernel constructs and forms a hierarchy of distinct operating environments. This model is crucial to code replacement and revision control. Supporting objects can be dynamically replaced without requiring notification of dependent objects. The broker interface hierarchy, and related software engineering techniques, also provide real-time guarantees that are superior to those

provided by related extensible systems. Brokers also act as the primary tools for all communications.

Communications are abstracted by message objects residing in memory. Thus, the memory model has a direct effect on communication performance. Chameleon employs a protected single address space (SAS) model to reduce communication overheads. In this design, multiple protection domains are mapped onto the same address space, and may share regions of memory. Chameleon is unique among research SAS operating systems in that it retains the file system, rather than using a distributed shared memory approach. Hierarchical resource management is applied to follow the run-time structure of the operating system. A minimal set of memory management primitives is also defined to allow a hierarchy of memory managers to work together. Hierarchical CPU management is used to allow different scheduling algorithms to co-exist.

Different forms of communication are identified and investigated. An object must be able to communicate with its operating environment for basic services, communicate with objects supported by the same operating environment, and communicate with objects supported by another operating environment, possibly on another machine. Further, all system events follow the same model to simplify the implementation of AbstractCPU and brokers. Communication primitives are then generalized to achieve consistency from an application's perspective with respect to communication, basic object storage, and object migration.

1.4 Performance of Chameleon

Comparisons with related work are given while Chameleon constructs are introduced and defined. These comparisons refer to the conceptual designs and are important in highlighting how Chameleon differs with related work, why particular

decisions in the design were taken, and identifying any advantages and disadvantages to those approaches.

Chameleon has been implemented as a stand-alone operating system. Various aspects of the system are profiled and quantitatively compared with other research and commodity operating systems. Three different techniques, namely instruction sequences and counts, simulated cycle counts, and actual hardware timings are used to prove and validate Chameleon's performance.

1.5 Organization of This Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 is devoted to a detailed description of the object model. Chapter 3 overviews related research that has had a strong influence on the design of Chameleon. Chapter 4 presents and defines concepts such as brokers, AbstractCPU, and the broker interface hierarchy that form the essential components of Chameleon architecture. Chapter 5 describes the protected single address space organization of the memory, as well as memory and CPU management in Chameleon. Inter-object communication in Chameleon is presented in Chapter 6. Chapter 7 discusses the implementation of Chameleon, details performance measurement techniques, and compares Chameleon's true performance against other operating systems. Chapter 7 also itemizes costs of individual portions of operations to show that the Chameleon architecture is a sound, efficient approach to operating system design. Chapter 8 summarizes and concludes the dissertation. It also outlines possible future work and further development of the operating system. Details of the operating system implementation, are given in the appendices.

Chapter 2

Object Model

Object-oriented designs have been the focus of much study since the early 1980's. In an 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 improving the reliability and maintainability of code, as well as exhibiting shorter development times. This is mainly due to their nature, which combines modularity, encapsulation, polymorphism, information hiding, and reusability [17].

Object orientation is a technique for structuring problems by dividing them into entities and relationships among them, thus reducing the complexity of the problems [18]. In non-object-oriented systems, different programming language elements are named with different terms such as types, variables, and procedures. In contrast, in pure object-oriented systems, all entities are objects that receive messages. Analytical descriptions are made of classes and objects throughout the design and implementation of the system. As such, the evolution of a system can be defined in a manner that is more consistent and understandable.

Another advantage of the object-oriented paradigm is that ideally the implementation details of an object can in principle 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. Inheritance allows one class definition to build upon, or specialize, or otherwise augment the properties and functionality of another class, and so is a powerful feature that provides for the reusability and extensibility of software components [17].

Initially, the use of object-oriented techniques was limited to the implementation step in the development of user applications using object-oriented programming languages such as Smalltalk [19] and C++ [20]. Object-oriented analysis techniques are now being used at earlier stages in the software development process, including the development of operating systems composed of and supporting objects. Operating systems control application processes and provide applications with the required resources and management facilities. This is not a traditional use of object-oriented constructs, primarily because operating systems must also control their own processing environment even though objects cannot provide their own infrastructure or execution environments. For example, some object must schedule a scheduler object. Hence, a minimum of two levels of infrastructures must exist. In some operating systems, including Chameleon, an object/meta-space hierarchy achieves this. This hierarchy is defined and examined later in this chapter.

There are variations in the definition of an object-oriented paradigm. For clarity, the important aspects and constructs of the paradigm as applied to Chameleon are defined.

2.1 Class and Object

A class can be thought of as a template that is used as a specification for creating new objects. This includes obvious components such as executable code called *methods*, data members associated with an object, and information such as how much memory an object will require. This information is shared among the objects, or *instances*, of the class. In Chameleon, a class is defined as static and immutable (i.e., constant) during run-time so that class duplication, object creation, and inheritance are easy to manage.

A class can be defined as a *subclass* of other classes, either by single or by multiple *inheritance*. A subclass inherits properties from its *superclass(es)*. Incremental

programming is encouraged using this superclass/subclass relationship, which defines the *class hierarchy*. Not all programming languages support multiple inheritance. Multiple inheritance is not essential for this work, but has proven particularly useful to simplify implementations for specific problems [21]. Where multiple inheritance is supported, constructs such as a C++ *virtual base class* (which prevents multiple instances of a base class object from occurring) become necessary.

Class definitions are static at run-time, though the class hierarchy is not necessarily so. The introduction of *compiled classes* at run-time, much like Java allows [22], to handle installations of new functionality is important to any extensible object-oriented system.

Each method for a class defines an action that can be performed by an object, which is an *instantiation* of the class. Constructor and destructor methods may be defined, and are invoked when an object is created and destroyed, respectively. A subclass can *override* a method implementation of its superclass. That is, a subclass's method implementation can replace or augment a superclass's implementation.

Some programming languages support the association of methods and data members with a class instead of with objects. (In C++, using the `static` modifier attains this.) Class methods and data members can often be used to simplify solutions to specific problems, where such data should be private to and not duplicated among all objects of a particular class.

A class can also be an object that is created and managed by a *metaclass* [15]. The metaclass defines the computation for classes. Smalltalk is an example of a system that follows this model. The discussion regarding the purpose and usefulness of a metaclass is not relevant to this work and is only mentioned for completeness.

An object is an instance, or instantiation of a class. Any number of objects can be instantiated from a single class. An object maintains its own storage for computation, which is to say that it is granted singular access to the data that comprises itself. Because an object encapsulates its data and actions, objects are similar to data types. An object is considered *elementary* if it contains no internal linked data structures or references (or pointers) to other objects. An object is considered *composite* if it contains references to other objects. Elementary objects may be linked together to form composite objects of arbitrary complexity [23].

Conceptually, an object can send and receive messages to/from other objects [17], even if the programming language does not implement communication in this manner. The object's class specifies the interface(s) by which an object can receive messages. It is often assumed that there is a one-to-one correspondence between methods and messages. An object's interfaces need not be defined in the same programming language as an object's class. A separate IDL (interface definition language) [21] is useful to allow objects implemented by different programming languages to easily communicate with each other. In this work, programming language-dependent communication optimizations have to be avoided and actual messages are often required.

2.1.1 Meta-Hierarchy

An object's class defines its computation. The management and the semantics of the computation (i.e., *how* actions are performed) are defined and handled by an object's *meta-objects*. An object's collection of meta-objects constitutes its *meta-space*. Other work [24] equates a meta-space to a computational environment or a virtual machine. A meta-object is also an object that has its own meta-objects, and is defined in detail later. This symmetry among objects, meta-objects, and meta-meta-objects (meta-objects of meta-objects) creates a *meta-hierarchy* that conceptually can be continued to an infinite

number of *meta-layers*. The term *meta* originates from language research, where the term *metadata* refers to data that represents the structural or computational aspects of an object [15]. Examples of metadata include the size of basic types (int, double, etc.) and programming language constructs such as constructors, destructors, and the implementation of virtual function pointer tables in C++. While metadata remain important in this work, the tasks handled by a meta-space are of more concern.

The concept of a meta-layer also originated in language research but has been adapted to operating system research. The term meta-layer is used here in a manner consistent with Apertos research [5], not PM research [10] where a meta-layer is also a meta-space. Although the terms *meta-object* and *meta-space* are well defined in language research, researchers are struggling with a clear definition in operating system work. This work improves these definitions later in this chapter.

2.2 Active Object

A *passive object* is the type of object just described, and that most programmers and researchers are familiar with because it is appropriate and sufficient for basic application programming. All regular C++ [20], Delphi [25], Smalltalk [19], Oberon [26], and Modula-3 [27] objects are passive objects. Some operating systems also use this model. For example, Oberon is a cooperative multi-tasking single-threaded environment [26] where all objects are passive and share the same thread of control. A passive object does not deal with synchronization or mutual exclusion problems and has no intrinsic thread of control associated with it. Method executions, consequently, simply run to completion. The programming language will guarantee any level of protection. This model has to be modified in many respects for a completely object-oriented operating system that aims for preemptive scheduling and interrupt-driven objects [28].

An important concept in the object model is the notion of a concurrent, or an *active object* [29]. Applications are programmed using passive objects – objects that are not inherently schedulable, and whose methods are simply invoked from actions originating from some outside source (user intervention, application start-up, etc.). Active objects, on the other hand, may spontaneously perform actions, even when no messages are sent to the object. When all entities in an operating system are ultimately objects, these objects must be schedulable and be able to asynchronously begin execution (e.g. to handle hardware interrupts).

An active object is an object that encapsulates local storage and methods (everything associated with a passive object) with a virtual processor, i.e. a thread of control. This virtual processor is referred to as a *locus of execution* [30] to avoid any preconceptions attached to the familiar terms *thread* and *process*. An active object also addresses synchronization problems. Simultaneous requests to an active object are synchronized and queued at the entry point of the object [24, 31]. Thus, in many respects, active objects are analogous to processes with well-defined communication ports and protocols. Memory protection and synchronization primitives are handled by the underlying operating system. A single locus of execution, combined with synchronization primitives, prevents an active object from becoming re-entrant.

There are no hard and fast guidelines used during the design of a component of an operating system to decide whether an object should be active or passive. This decision is ultimately made by the programmer who has some knowledge of how the object should be used. Further, there is conceptually no restriction that prevents an object from being instantiated as either (or simultaneously) active or passive.

2.2.1 Locus of Execution

A locus of execution is an object which can execute CPU instructions sequentially. It has its own stack and a location to store CPU registers, interrupt masks, etc., none of which are shared with other loci. It does not contain any additional information, such as scheduling priority or memory management. A locus of execution is in exactly one of many states at any given time. The minimal set of states is familiar to operating system programmers and is presented in [32] and is duplicated in Figure 2.1.

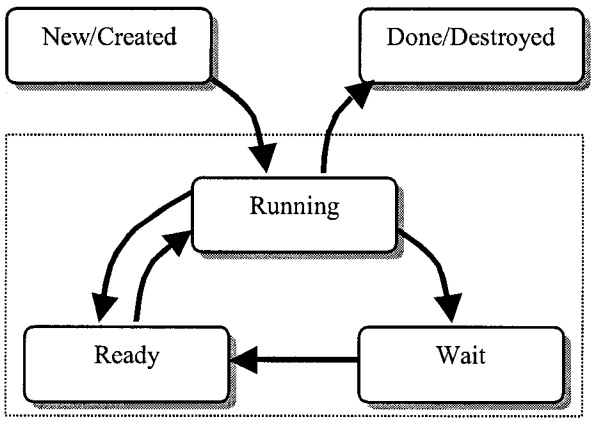


Figure 2.1 Minimum Set of States of a Locus of Execution

A locus of execution cannot be shared among active objects. It is created and destroyed with the single active object with which it is associated. The same object may, however, reuse its locus of execution for many method executions.

A locus of execution may inherit properties (such as ranges of accessible memory addresses) from a parent locus of execution. The relationship between a parent locus and a child locus is somewhat analogous to the relationship between a process and a thread in a traditional system and should not be confused with class inheritance.

A locus of execution may be extended via class inheritance. For example, a possible extension to the minimal locus of execution may be to associate real-time

constraints, or even a simple scheduling priority value. Extensions may also define new states for a locus of execution.

Other work has experimented with multiple loci per object [33]. This avenue of research is not pursued in this work to avoid the re-introduction of synchronization issues within the object, and because one locus per object is sufficient and appropriate for the Chameleon model.

Inter active object communication is actually implemented using message objects, so that entities external to the compiler (the scheduler, for example) can manage the active objects appropriately. Presently, the programmer is required to write all message classes, but these classes inherit from the predefined message class to take care of some mundane tasks.

2.3 Meta-Object

The term meta-object is always relative to another object, called the *base-level object* [34]. With respect to operating system research, meta-objects are generally defined as having the following properties [5, 10]:

- An object is created and destroyed by its meta-objects.
- The execution semantics of an object's methods are conceptually defined and handled by its meta-objects. In other words, while the class defines the actions to be performed in the method, the meta-objects define *how* such actions are to be invoked and/or performed. For example, a meta-object may choose to apply a software implementation of IEEE floating point math rather than allow an active object to use built-in (hardware) floating point math instructions in order to obtain a different degree of accuracy.

- The communication semantics among objects are defined by their meta-objects. Meta-objects are responsible for providing an object with local storage having any desired properties, such as persistent storage.

At any time, a meta-object knows the status of its base-level objects, and conversely, an object may request the status of its meta-object. This rule is analogous to an application program querying its underlying operating system for internal system state information, and a debugger knowing the state of a program being monitored.

The class hierarchy is independent of any object's meta-hierarchy. Since a class is a static template in Chameleon, it is a programming and compile-time entity. Objects, meta-spaces, and the meta-hierarchy are dynamically created and destroyed at run-time. Therefore, a class is independent of the level in the meta-hierarchy where an object is instantiated and placed. This is illustrated in Figure 2.2. The *meta* dependencies among objects and their meta-spaces and between classes and the meta-class are shown. In the

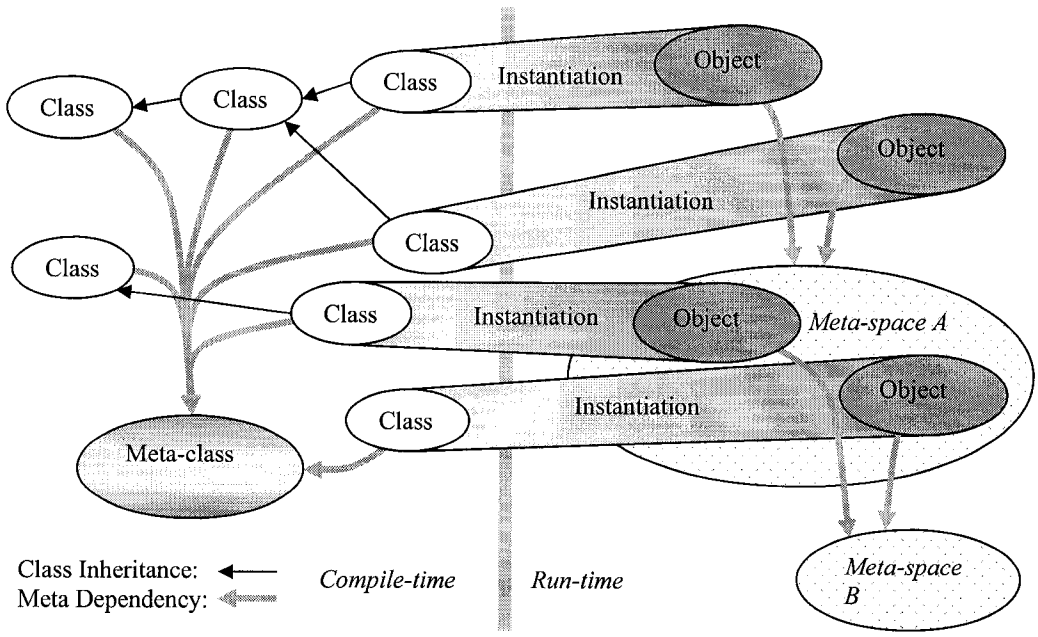


Figure 2.2 Entities in the Object-Oriented Paradigm

figure, objects in meta-space *A* are meta-objects to the objects supported by meta-space *A*. Objects in this meta-space are supported by meta-space *B*, so are considered base-level to meta-space *B*. Meta-objects are not instances of the meta-class but rather instances of classes. Multiple inheritance is not illustrated.

An object is not concerned with the implementation details of either its metadata or its meta-objects or meta-space. However, what is of concern is the means by which its meta-level data and policy are accessed and possibly modified, since the contents of the metadata and the policies of a meta-space are important to its run-time behaviour. All systems exhibit some form and type of this functionality. The object-oriented paradigm simply refines this mechanism so that it can be identified and exploited.

2.3.1 Meta-objects vs. Service Objects

This work refines the above definition of a meta-object. Previous work left this definition so generic that objects providing services to other objects could be called meta-objects, a problem arising from the translation from language to system research. A perfect example is a specific scheduler object. A scheduler object implements a *service* for one or more objects. Such an object does not have to be aware of the make-up or internal composition of the objects it serves, it only has to be aware of the requirements of the objects that it supports.

This work does not consider a service object to be *meta* to the objects that it supports. A meta-object provides the binding and the logic, and thus provides the associations among service objects and base-level, or application objects. For a meta-object to act as this glue, it may have to be aware of the internal composition or properties of an application object. An example of this is the entity that is aware of the API (application programming interface) that the base-level object expects to use, to ensure that the appropriate service objects are available. Conversely, service objects are

more akin to handling, for example, physical faults and interrupts, such as keystroke events. Meta-object behaviour is relative and causally connected to base-level computation [35]. Consequently meta-objects are not applicable to handling hardware interrupts because there is no associated base-level object for the interrupt event.

This distinction is compatible with research studying meta-objects in languages where meta-objects and the corresponding subsystems, being an operating system kernel, or service objects, are together considered a meta-layer [36]. [36] only accounts for static binding between the base-level and meta-layer objects. This work examines dynamic binding.

Other work [37] outlines another compatible definition, and describes meta-objects as only *module specifications* that are extension programs whose execution results in the generation of executable fragments. In Chameleon, meta-object execution may not generate executable fragments, but can collect and associate together pre-generated executable fragments encapsulated as service objects.

Further, an examination of the definition of a meta-object protocol (MOP) [36] shows that meta-objects should be distinct from service objects. The essence of a MOP is to give the user object the ability to adjust an implementation to suit its particular needs. A MOP is not intended to handle keystrokes, or make requests to or communicate with a particular implementation of a service, i.e. a service object.

This distinction between meta-objects and service objects requires a modification to the definition of a meta-space. In Chameleon, a meta-space remains an abstract collection of objects. However, rather than comprising of only meta-objects, a meta-space is comprises:

- one meta-object, called a *broker*, that provides the logic to associate service objects to application objects, and to provide the interface used by application objects to gain access to their services; and
- a collection of (one or more) service objects that implement the services required by the application, or base-level objects.

This definition ensures that a meta-space remains analogous to a virtual machine, or a complete operating environment for a base-level object. Note that no rule specifying whether meta-objects or service objects are active or passive, is made.

2.4 Object Model Conclusions

This section has defined and refined the definitions of various abstractions: classes, objects, meta-objects, service objects, meta-spaces, etc. These abstractions are not overtly different from current abstractions in operating system and general application design. For example, an active object abstraction in many ways correlates to the traditional process model. There are also similarities between a simple thread and a locus of execution. However, there are some distinct differences. For example, there is no single identifiable counterpart for a meta-object in a traditional operating system. These distinctions are used to identify, define, and explain the new Chameleon operating system concepts. Furthermore, it will become apparent that current abstractions, such as micro-kernel and process, are insufficient and limiting to describe or visualize this new model.

An effort has been made to use existing abstractions whenever possible, and refine and correct definitions, such as meta-object and meta-space, as necessary. In doing so, this work has neither invalidated nor compromised previous work or definitions. Finally, these definitions have not become so binding as to limit implementation to a specific object-oriented programming language.

Chapter 3

Related Work

This chapter reviews operating system research related to this study. Each operating system discussed was designed with extensibility and performance in mind, but with different solutions. Each is overviewed and essential elements in their designs will be compared against Chameleon.

3.1 Apertos

The Apertos project [5, 38, 39] from 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. *Base-level* objects are analogous to applications and are supported by *meta-objects* that comprise meta-spaces. A *meta-space* is analogous to an application support layer and is expected to be optimal for the objects that the meta-space supports. Meta-objects are in turn supported by their own meta-spaces, producing a *meta-hierarchy*. In theory, this meta-hierarchy can be infinitely deep but, in practice, it is limited to three layers. Two meta-spaces exist on the lowest meta-layer to support each other so that Apertos will follow a pure meta definition: given any layer in a meta-hierarchy, there is another layer *meta* to it. The circular dependency at the base of the meta-hierarchy is unique to the Apertos model, and is illustrated in Figure 3.1. The object model is used primarily for two reasons: to provide a uniform perspective to users and programmers alike, and to simplify object replacement.

Apertos was also the first operating system design based on *reflective computing* [24]. The operation by which an object views or communicates with its meta-space is

called *reflection* [15, 38]. All objects communicate with their meta-objects via their *reflectors*, which are special meta-objects that forward incoming requests to appropriate meta-objects [40]. Objects can share reflectors, and hence share meta-objects. Figure 3.1 presents a conceptual overview of the Apertos structure.

Various communication models are applicable to Apertos (asynchronous messages, group sends, spontaneous activity, etc.). In practice, however, the use of only synchronous or blocking calls, combined with the fact that all active objects in Apertos are single threaded and non-preemptable [16], reduces Apertos to a sequential object-oriented model as defined in [41]. Hence, no functionality would be lost if the system were reduced to a single thread of control.

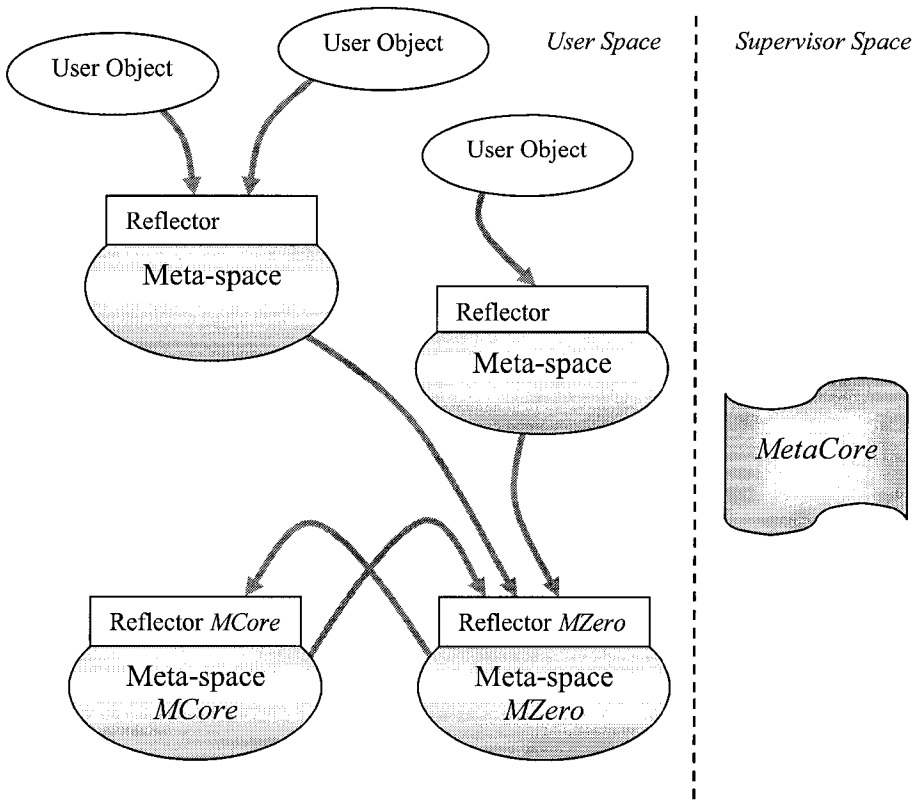


Figure 3.1 Apertos Layout

An Apertos reflector is intended to be an active object that abstracts a collection of meta-objects comprising a meta-space. In practice, however, an Apertos reflector is an active object that defines a meta-space in its entirety, a major withdrawal from the intent of the design. This decision was made to avoid a serious impact on communication performance. Even with this compromise, Apertos suffers from poor performance [16]. Combining reflectors and meta-spaces impacted the overall flexibility of Apertos since the means for an object to communicate with a reflector is rigid. Since reflectors are not abstract interfaces to meta-spaces, only a single meta-object protocol (MOP) can be supported [42]. Object migration to other meta-spaces is limited to a subset of reflectors supporting the identical interface.

Since reflectors are active objects, they must be explicitly scheduled. This requires the invocation of the meta-space supporting the reflector to ensure consistency of the system model. Consequently, every meta-space ultimately supporting an active object is invoked during any form of communication. This fundamental design decision alone severely impacted communication performance [16].

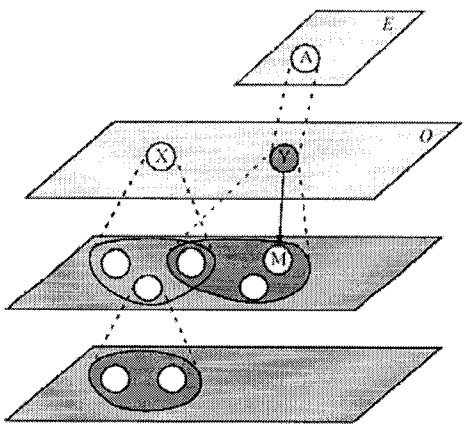
Apertos is flawed primarily because it has unintentionally mixed new concepts such as meta-object with traditional concepts such as a micro-kernel. Details of the commercial successor to Apertos, called Aperios [43], are limited because the system is proprietary to Sony. However, Aperios is believed to have retreated considerably from the design of Apertos in order to obtain performance adequate for a video set-top box application. Despite these shortcomings, Apertos must be considered a crucial step in object-oriented, meta-space operating system designs, and thus towards the Chameleon model.

3.2 PM Operating System

The PM distributed object-oriented operating system [10, 44] was a project at the Friedrich-Alexander University, Germany, 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 programming 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.

In PM, any application can build operating system components on top of existing abstractions for services and hardware. These abstractions, called *meta-layers*, require interfaces between an application layer and the meta-system. This relationship directly corresponds to the object/meta-space relationship using reflectors in Apertos.

Virtual address spaces are called *object spaces*. Although, in theory, objects can be mapped into several object spaces simultaneously, it is not discussed exactly how this would be performed. While Apertos emphasizes the ability of objects to migrate between



enhanced application layer ***E***
object **A** is implemented within Object Model ***E***

application layer ***O***
objects **X, Y** are implemented within object model ***O***
object **Y** implements an enhanced abstraction for object model ***E***

meta-layer which provides a complete implementation of all abstractions of object model ***O***
interface of object **M** is accessible for application object **Y**

meta-meta-layer which provides an implementation for the meta-layer

Figure 3.2 PM System Architecture

meta-spaces, PM does not. When an object is placed in one meta-layer in PM, it remains there for the duration of its existence. No reason for this design restriction appears to be given.

Just as other systems do, PM strives to separate hardware architecture-dependent from architecture-independent code [44]. Chameleon follows a similar approach to minimize the size of low-level hardware abstractions.

3.3 Oberon

Oberon is both an object-oriented programming language and an operating system, and was designed by Niklaus Wirth and Jürg Gutknecht at ETH Zürich. A primary goal of the Oberon project was to design a system that can easily evolve over time, and that is very flexible and extensible, as it is impossible to foresee all of the requirements that will be imposed on the system in the future. Low memory usage is another goal of Oberon. Security is not a goal of the Oberon system.

Oberon is an open system consisting of a set of modules sharing the same address space. No memory protection is provided among objects, and the system depends on the strong type checking and other features provided by the programming language for security. User-written modules can be added at run-time to extend the functionality of the system, or to replace system modules.

The nucleus, or the portion of Oberon that cannot be dynamically replaced, is minimized. All entities of the system are passive objects and operate in a cooperative multi-tasking, single threaded environment [26]. A *central loop* is used to coordinate the actions of all objects. Event sources such as the keyboard and mouse are polled as an alternative to being driven by interrupts to remove synchronization issues. Work in [28] introduced preemption and synchronization solutions to Oberon.

3.4 Spin

Bershad et al. [12, 45, 46] have developed an extensible operating system called Spin. Extensibility in Spin is based on intercepting intra-kernel procedure calls and funnelling them to dynamically installable handles called *spindles*. This solution was chosen because some analysis has shown that a major performance bottleneck in current operating systems arises from the many context switches between kernel and user spaces [13].

The approach used in Spin is that any procedure call, including those inside the kernel, can be considered an event. All kernel code and handlers are written in Modula-3 and rely on the safety features of the programming language and compiler to verify the integrity of the code. Procedures in Modula-3, and thus events in Spin, have to belong to

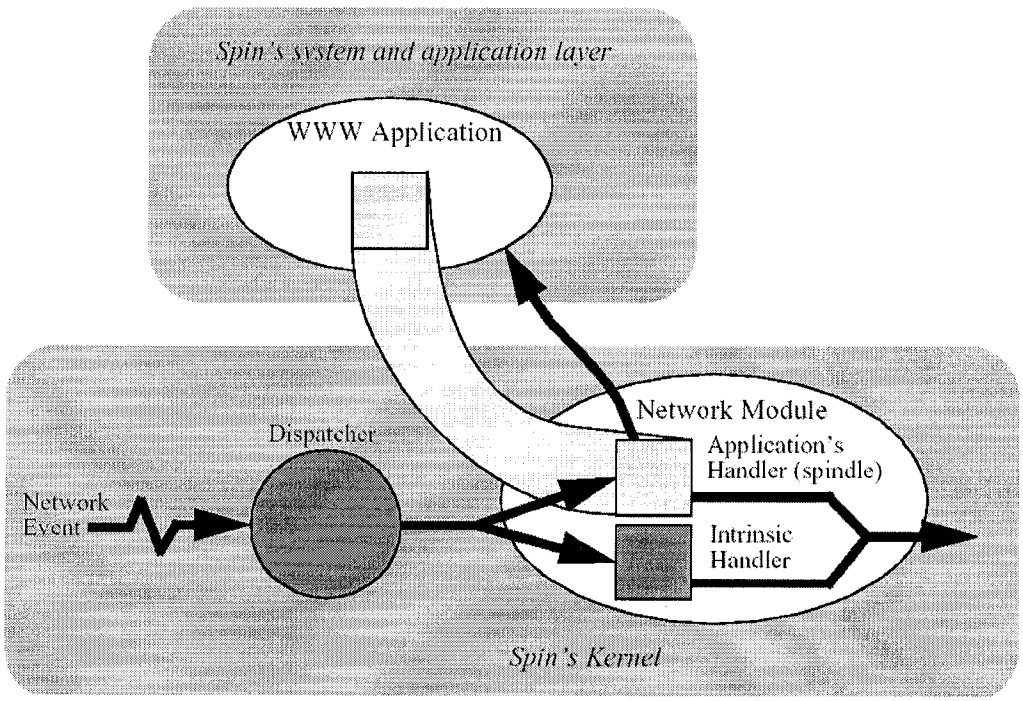


Figure 3.3 Example Spin Event Handling

a module. The module which provides the default or initial handler (called the *intrinsic handler*) for the event can accept or deny any request to install an extra handler to an event [46].

Handlers can have *guards* associated with them, can be ordered, and are invoked using the system-wide dispatcher. A guard is a very small function that does an initial check to determine whether the handler, which is intended to contain more logic, should be invoked. Both the owner module and the modules supplying extension code can provide guards for handlers [45].

When an event occurs, the handlers are invoked either as procedure calls or as kernel threads (which explains why guards exist). The actions that these handlers perform may range from signalling an application to implementing a complete Internet daemon server. Finally, handlers must return in a timely manner, or the handler's execution will be aborted to allow Spin's kernel to continue. The result of Spin's approach is a kernel that is extensible by installing application-specific guards and handlers.

Figure 3.3 shows a WWW application running on Spin with an event handler (spindle) installed into the kernel for a network event. Upon the occurrence of a network event, the dispatcher will signal each handler in succession that is associated with this event. Here, the dispatcher signals the intrinsic handler as well as the application's handler. In this case, the application's handler signals its application that more processing is necessary and kernel execution resumes with the completed intrinsic handler. Guards are not shown.

What is not discussed is whether the dispatcher exports any events that can be extended upon. In a system such as Spin, this may seem obvious, but the designers may have decided against this due to the potential performance impact, the possibility of code

becoming re-entrant, and the security risk. As well, there is no discussion regarding whether the dispatcher will watch for infinite loops through the dispatcher mechanism, which could be extremely difficult to identify and correctly restore to a stable state.

3.5 Vino

Vino [47, 48] is an interesting operating system from Harvard University that is comparable to Spin. Although the systems are very similar in many areas, Spin emphasizes extensibility where Vino emphasizes code reuse [13].

In contrast to Spin's dependence on programming language safety features, Vino uses *sandboxing* to prevent its extensions, called *grafts* [14] (normally written in C++), from failing with catastrophic effects, or from taking over the machine. Sandboxing is a technique used to monitor every memory access to try to prevent and/or catch erroneous code before any damage can occur. [13, 14] give measurements to show that sandboxing presents a lower overhead than interpreted languages as a way to guarantee safety, and conclude that any costs of sandboxing are outweighed by the gains in extensibility.

Every graft is invoked as a transaction so that in the event of a failure, the kernel can revert to a previous, stable state to continue execution. As with Spin, extension code can be invoked as either a procedure call or as a kernel thread, a decision that can be difficult and even arbitrary at times. The use of transactions to identify and correct any infinite loop through the message dispatcher (which Spin is susceptible to) should be possible. That is, it should be possible to identify a problem where a handler erroneously continues to send itself the identical message.

There exists an *inner kernel* and a preemptable kernel space in Vino. The inner kernel has to be protected from extensions that may fail, and makes global decisions affecting the entire system. Extensions reside in the preemptable kernel space and make local (or application-specific) decisions on resource usage [13, 48]. In fact, kernel

extensions running as threads are considered user-level processes that happen to run in the kernel's address space [47]. In the interests of performance, code that would have been processes running atop the kernel in a traditional system (e.g. internet daemons) are instead placed in extensions. The basis for this decision is research that has identified context switching overhead to be the bottleneck in micro-kernel systems.

As with Spin, Vino is very good at addressing the problem of evolutionary changes to the kernel/operating system boundaries to match changing application requirements and operating technologies and hardware. Their designs make them ideal for experiments and easy design tweaking for optimal performance. However, these gains come at the expense of relaxing previously well-defined terms (such as *kernel*) and operating system structuring rules (e.g., the clear line drawn between the application and the operating system). In fact, Vino has introduced another level of abstraction and complexity (inner kernel vs. preemptable kernel, kernel threads treated as user processes in kernel space, etc.).

Other work [49] questions the validity of the Vino and Spin works because it is unclear to what extent that the cited performance gains of the presented benchmarks are due to extensibility, or are due to optimizations that could be applied to any operating system.

3.6 Fluke

Fluke is a software-based architecture that allows virtual machines to be implemented efficiently atop a micro-kernel [1, 50]. Here, virtual machines are used for enhancing operating system modularity, flexibility, and extensibility rather than simply hardware multiplexing. Thus, operating system features are divided among virtual machines that can then be specialized for the applications that they are intended to support.

Virtual machines can be "stacked" upon each other, similar to meta-spaces in Apertos. Measurement numbers show an execution cost of 0-35% per virtual machine layer, which is very promising in light of Apertos' heavy overhead. Fluke succeeds in this area because the exponential slowdown in traditional recursive machines, which is the same in layered meta-space systems, was properly identified and circumvented through message passing short-cuts.

Many ideas in the Fluke system map directly to models in the *meta* structure: hierarchical resource management, shared interfaces, and the notion that a virtual machine completely simulates the environment needed for the layer it supports (which may include other virtual machines). However, Fluke does not allow applications to roam to other virtual machines, a limitation in the flexibility in its design.

Fluke is not an inherently object-oriented design, even if it does take advantage of object-oriented techniques such as interface inheritance. This is regarded as an advantage to support legacy UNIX applications.

3.7 Synthetix

The Synthetix operating system [51] is a very practical approach to automatically provide specialized implementations of various services and to allow run-time swapping of these services. For example, in a file system, an open call no longer returns a file handle but rather returns a handle to a section of code optimized for performing operations on that file. The system reserves the right to change the code that the handle references to allow the implementation to be improved upon over time.

Synthetix' limitation is that it does not provide the application with a means of modifying the default implementations [52]. The application must expect the operating system to have at least one implementation of a service that will provide good performance, because the application itself cannot provide such an implementation. The

authors argue that this is a better solution because reliability and security hasn't been compromised. As all variants are known *a priori*, it is also known which combinations of variants are valid [51]. Synthetix, therefore, is adaptable but not extensible.

The Caching Kernel is similar to Synthetix, and can load specific kernel modules into a "cache" in kernel space, where the type of services are pre-determined [53]. Types of services are limited to kernel codes, address spaces, memory spaces, and processes. There are no user-definable services allowed.

3.8 Kea

Kea, developed at the University of British Columbia, is an operating system kernel designed to maximize flexibility and performance [52]. Rather than injecting code into the kernel as Spin does, Kea follows a thread model similar to Spring's [52, 54], where threads are allowed to cross protection domains via *portals*. By allowing threads to jump across (potentially many) virtual address spaces, the performance hit from switching among processes and kernel mode occurs less frequently. However, since Kea's threads are seldom associated with any single entity, mutual exclusion and recovery from failed threads become issues. If a failed thread's path across protection domains was not closely monitored, then resources could easily be lost and objects could be rendered unusable (i.e., put into an invalid state). Checkpointing such a thread in a highly concurrent system may be difficult when protection domains are crossed, since dependencies among threads are not explicit. Nonetheless, there is merit to the design; a variation on this technique was chosen for Microsoft's COM threading models [21].

3.9 ES-Kit

ES-Kit is a set of software and hardware building blocks that may be readily assembled to produce a heterogeneous, object-oriented distributed system [55]. 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, based on its dynamic state, to overrule the guarantee provided by the underlying system that method execution is strictly in the order of message arrival to 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.

A natural extension of method locking would be *message forwarding*. If an object manages a printer that happens to break down, then that object may wish to have all pending print jobs (message requests) forwarded to a compatible object managing another printer. A degree of fault tolerance is realized.

3.10 Conclusions from Related Work

The works reviewed above have directly influenced the design of Chameleon. There are positive and negative aspects to every design, and many important conclusions can be drawn, collectively, from these efforts.

Granularity is a problem that is continually addressed and re-addressed by most of these designs. In Apertos and PM, the problem is where to draw the boundaries between meta-objects; Fluke's problem is to determine the optimal contents of a single virtual machine. For Spin and Vino, the decision is whether to invoke a kernel extension (a

spindle or a graft) as a thread or a procedure call. A primary problem is that the designer does not have a crystal ball showing how the objects, or kernel events, will be extended upon. The approach used in Chameleon is to defer these decisions to a later time, when this information is known.

Another problem similar to granularity but involving security and correctness of operation, is also present in Spin and Vino. The kernel extension techniques used in these operating systems are analogous to event handling in Mac-OS and MS-Windows GUIs where applications can listen and act on only the events that the application is interested in. The glaring assumptions here are that every event that an application may want to monitor is available to it, and that the application programmer is able to determine exactly which event is the appropriate one to extend upon. In fact, [47] admits "there's a continuing battle between expressive power and simplicity in designing a safe graft interface". This issue is compounded by the fact that while an application programmer can experiment with the appropriate events to use in MS-Windows and Mac-OS within the confines of a message queue that is private to the application, the message dispatchers in Spin and Vino are in kernel space and "shared" among the entire system! Application programmers developing software for either the Mac-OS or MS-Windows GUI's will attest to the degree of difficulty in determining the appropriate messages to monitor, and the appropriate actions to perform when events do occur, so compromising the correctness of the system *even within the confines of sandboxing or programming language guarantees* is inevitable. A better solution is to avoid the temptation of solving this entire problem at once by placing everything inside the kernel. Operations not required to execute in supervisor mode should be left in user space for security and reliability, and interface definitions should be carefully designed and concise.

All the reviewed systems have used interesting techniques to achieve performance that is considered acceptable by their designers. Apertos combined reflectors with meta-

objects to avoid context switches; Spin and Vino allow applications to be implemented as kernel threads, and Kea allows threads to cross protection boundaries. Context switching overhead is uniformly identified as the bottleneck in micro-kernel systems. However, other work [56] challenges this premise with hard numbers. From these facts, we can conclude that if kernel threads are sufficient for performance, but processes incur too much overhead, then the traditional thread/process model must be re-examined. The solution put forward in this work is to replace the thread/process model with the concept of a *locus of execution.*

Chapter 4

Chameleon Structuring Concepts

The Chameleon operating system architecture is designed around a series of objects called *brokers*. These brokers are designed using the object model defined in Chapter 2. Brokers, with supporting objects, implement much of the functionality found in traditional kernels. Brokers' duties are also similar to Apertos' reflectors in functionality, although the design is entirely different. The design and inefficiencies of Apertos' reflectors were overviewed in Section 3.1. This chapter begins with a review of kernel designs to serve as a motivation for the broker design, and to place discussions of brokers into proper context. After brokers are introduced and discussed, a comparison against Apertos reflectors is given.

4.1 Kernel, Micro-Kernel, and Abstractions

There are many definitions of kernels and micro-kernels [57]. The only common theme apparently existing among all definitions is that a kernel consists of only the part of the operating system that is mandatory and common to all other software [51], and a micro-kernel design minimizes this part. A module is tolerated within the micro-kernel only if moving it outside (and permitting competing implementations to exist) would prevent the implementation of the system's required functionality. In contrast, a complete operating system provides general, or common services to applications and alternative services may exist [11]. Two representative micro-kernel systems are Mach [9] and of Windows NT based systems [58].

The primary purpose of a kernel is to provide the most basic, primitive functions that act as a foundation for different operating system extensions. In theory, this separation of functionality from the rest of the system gives an operating system the ability to provide multiple environments for applications. In practice, however, kernels are unwieldy and difficult to maintain and extend upon, and consequently cannot easily adapt to changing system requirements. Micro-kernels tend to be relatively inefficient and difficult to re-use because more context switches are involved, and policies that are intrinsic to their designs directly impact the subsequent operating system designs. As more features are added to satisfy new application requirements, the systems become more difficult to maintain.

Figure 4.1 presents a visual representation of how some operating system structures distribute responsibilities. Still, opinions differ in what functionality should reside in a micro-kernel or kernel, and micro-kernel designs that may in fact be comprised of modules, often combine hardware abstraction (such as virtual memory page table handling) with low-level policy (such as LRU vs. MRU virtual memory page election policies) [59]. As examples:

- Some definitions for a micro-kernel include device drivers [60, 11].
- Spin [12] allows servers to execute as part of an extensible kernel, citing kernel threads as being more efficient than user processes as motivation.
- Exokernel implements virtual memory and IPC abstractions in application-level libraries [57], and attains good performance.

As a result, meaningful performance measures and comparisons become difficult to perform.

The identified problems are therefore mainly due to the lack of a consistent model and the difficulty in maintaining and extending the kernel and operating system. To address these issues, this work moves the line of abstraction normally drawn between a

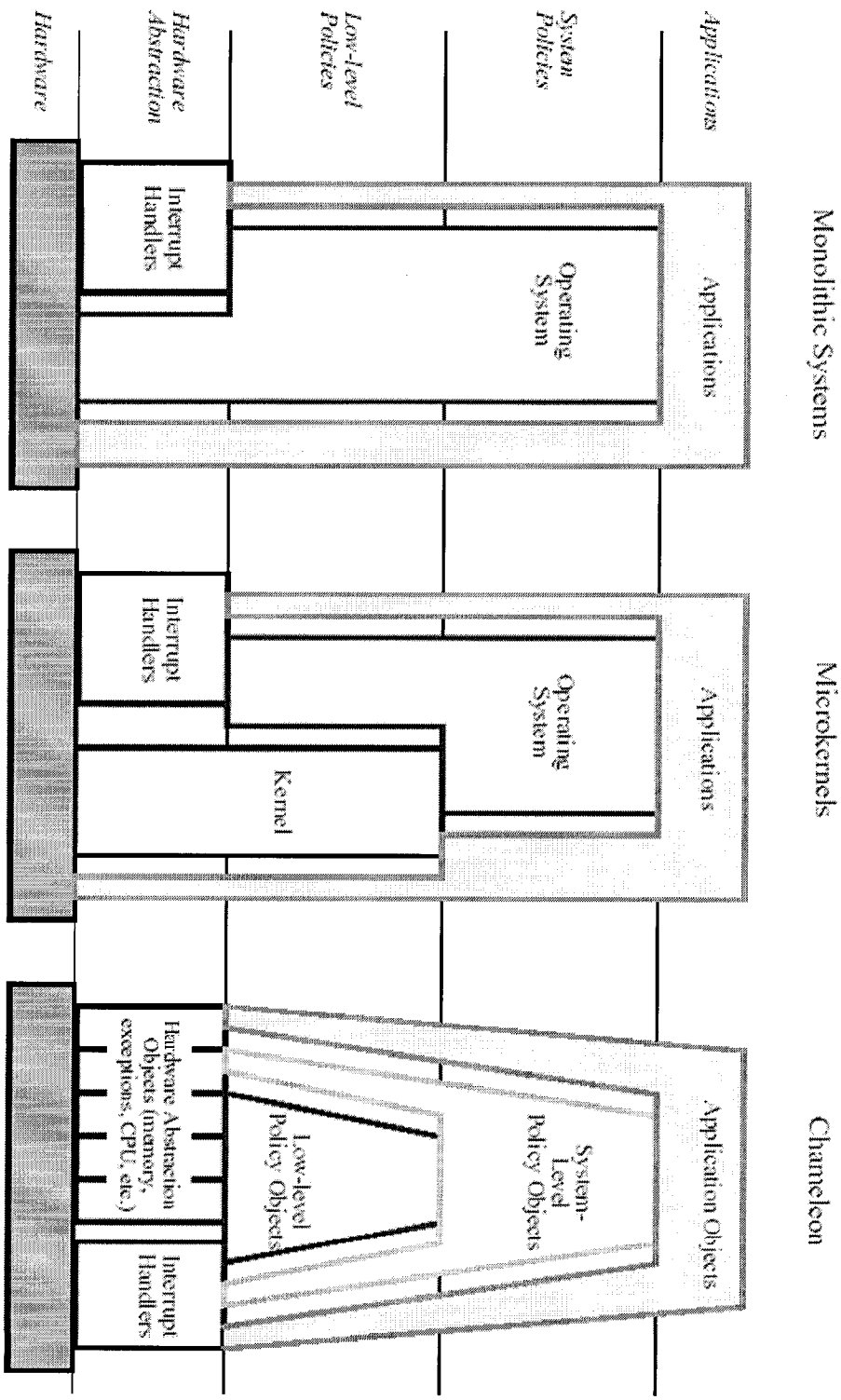


Figure 4.1 Conceptual Views of Operating System Structur es

micro-kernel and the operating system, to one that is between low-level objects which only abstract hardware and system-level objects that implement policies. This modification is illustrated in Figure 4.1. Previous work, such as Oberon [26] has proven that the performance penalty incurred for this line of abstraction can be negligible. However, Oberon does not support dynamic swapping of application-specific objects [61], which is a fundamental aim in this work.

Hardware abstraction objects are special service objects that support system-level objects by performing tasks such as context switching, I/O management, and interrupt handling. One object exists for each hardware resource to easily mimic any hardware configuration changes (such as modems added at run-time, or CPU cards hot-swapped). Objects implementing system policies are also service objects as defined in Chapter 2, and for example, may implement a particular scheduling algorithm.

Policy and hardware abstraction objects must interact closely and efficiently to address performance concerns. For example, a hardware abstraction object would be invoked for the arrival of a network packet and immediately record this event to be handled by an appropriate system service object. Two policy objects may use the same network hardware abstraction object but implement substantially different policies: multimedia applications need timely delivery and will accept errors in transmission whereas database applications can sacrifice real-time response for correctness of the data. This division of functionality provides a clean solution to allow both databases and multimedia applications to use the network. Hardware abstraction objects and policy objects together define much of the functionality found in kernels and their operating systems; the difference is in how these entities and objects are defined.

This change from a traditional design is important to the Chameleon design and has many advantages:

- objects will be smaller and should be easier to maintain and prove correct;
- an object-oriented design helps enforce well-defined interfaces and divisions of functionality simplify revision control;
- higher-level objects will not always be subject to immutable micro-kernel policy decisions; and
- migration of the operating system to different platforms should be easier.

These advantages are discussed as the broker interface design is defined throughout this chapter. The implementation of Chameleon is empirical proof that these claims hold true. However, this change requires two previously defined hierarchies to be carefully leveraged.

- The class hierarchy is important so that objects sharing interface definitions can be dynamically replaced with one another. The class hierarchy defines sub-class/super-class relationships. Refer to Figure 2.2. Method up-calls from a (base) super-class to a (derived) sub-class allow a generic and even incomplete super-class to exist. Method down-calls from a sub-class to a super-class allow the application of incremental programming techniques in object-oriented designs. Class hierarchy relationships are static at run-time.
- The meta-hierarchy is used to define run-time dependencies and relationships among objects and their support layers. The meta-hierarchy is independent of any class hierarchies and creates a type of a dependency graph among objects and environments. Here, a meta-hierarchy is comprised of application (base-level), meta-spaces supporting objects, and meta-spaces supporting meta-spaces. The meta-hierarchy is strictly a run-time construct, and is allowed and expected to change over time.

Interfaces to classes must be carefully designed. A hardware abstraction object's interface will define the limits of a system's extensibility because all higher level abstractions necessarily depend on it [62]. Thus, policy must be limited to higher level objects. The micro-kernel / operating system model doesn't lend itself to easy extension to support completely new problem domains for this reason [42], as virtually every micro-kernel and kernel implements policies.

4.2 Interfaces and Dictionaries

Simply calling every entity in the system an object is insufficient. A well designed object-oriented system makes full use of the separation of *interface* from *implementation*, even if only at the conceptual level. C++ provides this type of functionality through virtual functions [20]. Java [22] provides an interface construct, separate from a class construct. Smalltalk [19] is interpreted so that interfaces and implementations can evolve at run-time. These are features of programming languages. Chameleon makes strong use of interfaces in its most basic important construct: the *broker*, where all brokers inherit from a common interface.

An object's interface typically defines its *dictionary*. A dictionary is the set of messages that the object accepts, and each message in the dictionary represents a method call. This model is straightforward and easy to understand, but is insufficient for Chameleon brokers and meta-spaces with different policies and implementations. Using this model, the broker interface would have to support all possible messages that all brokers (present and future) could accept. This is obviously unreasonable but clearly illustrates a problem that is inadequately addressed by almost all major current extensible operating system project. Simply stated, it seems impossible to define an interface that is adequately expressive for *all* perceived scenarios but remains compact and concise enough to make it usable, safe, flexible, understandable, and implementation and

hardware independent [47]. This problem is particularly evident when dealing with the introduction of relatively new problem domains, such as those in multimedia applications.

Chameleon introduces a completely different and novel approach:

- All messages that can be introduced to the system must inherit from a common class: *Message*. The common message class has two members: a sender and a receiver. The receiver value can be undefined when the active object introduces a message to the system (to make a request to its meta-space where the implementation is unknown), but the sender value is needed to identify the active object introducing the message.
- The supporting system provides an exported interface that is used as a means for any active object to introduce a message to the system (i.e., send a message to its meta-space). This exported interface is small: one method with exactly one parameter: an object whose class inherits from *Message*.
- Any broker's dictionary is composed of classes inheriting from *Message* that it can accept. A broker must accept the set of messages abstracting hardware (e.g. floating point error) and inter-broker events (e.g. name look-up), but is free and expected to define the extensions to that set that defines its meta-space's API. There is no assumption that a broker's dictionary (set of messages that it will accept or understand) is static at run-time.
- A class called *AbstractCPU* is present to route messages from the exported interface (containing the single method, introduced above) to the broker interface. By design, active objects have no direct access to either brokers or the broker interface so that brokers can be dynamically replaced, and dictionaries can vary among brokers. The broker interface is common to all brokers and is necessarily different from a broker's dictionary. AbstractCPU's algorithm to route these

messages only understands the common message class; it is not concerned with the particular class of the message object that it is handling due to the class hierarchy and the nature of the object-oriented paradigm.

The broker interface is unique and must be carefully designed. From one perspective, it defines operations that are *meta* to basic communication. Base-level objects must use the base-level exported interface which contains the single exported method. Brokers are *meta-objects* and AbstractCPU is the mechanism that works between the base-level exported interface and the meta-level broker interface. This layer of abstraction is paramount to achieving flexibility that isn't present in micro-kernel designs, and to realizing a level of performance that wasn't possible in Apertos, which used meta-objects but worked within the confines of a microkernel model. However, the broker must eventually determine appropriate actions based on the class and contents of a message in its dictionary. This resolution from the broker interface to the contents of the broker's dictionary is left to the broker and is expected to be implemented as a straightforward switch statement.

4.3 AbstractCPU

The name AbstractCPU is descriptive: it is an abstract class that represents the CPU and its privileged, hardware-specific operations for the operating system and applications. A single AbstractCPU object exists in a Chameleon system. It is programmed to perform elementary context switches, act upon interrupts and exceptions, and handle messages introduced by active objects. Given its duties, AbstractCPU is bound very tightly to the hardware. It is not considered dynamically replaceable due to the same hardware constraints and reasons that make micro-kernels hard to change at run-time. However, there is no need to dynamically replace AbstractCPU because there

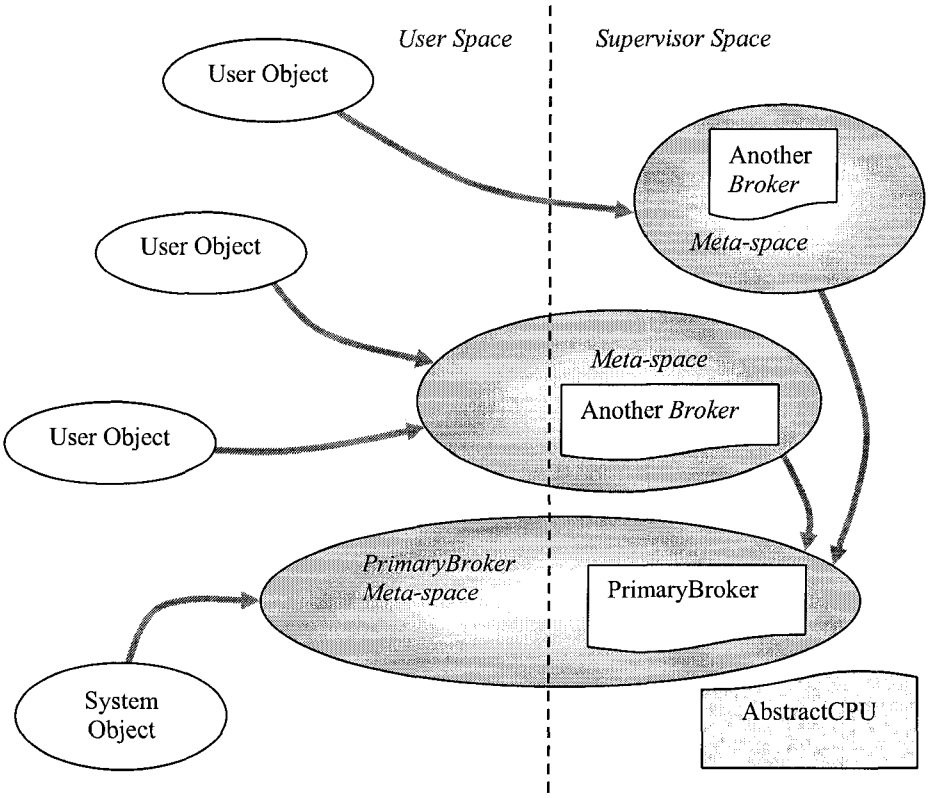


Figure 4.2: Conceptual Layout of the Chameleon Architecture

is no operating system policy implicit in its algorithm; it only supports brokers implementing policies.

4.3.1 Tasks of AbstractCPU

All communications between an active object and its broker/meta-space are routed through AbstractCPU, from the exported interface and then via the broker interface. AbstractCPU also provides an environment for all broker executions, as well as a mechanism for direct inter-broker communication. This environment includes the execution stack shared by all brokers. Brokers are not considered schedulable entities because AbstractCPU invokes them directly via the broker interface. Consequently, all brokers execute in supervisor mode and are passive, not active objects. A conceptual

layout of brokers and AbstractCPU in the Chameleon architecture is given in Figure 4.2. Each meta-space must have a broker placed in supervisor space, and may include service objects executing in user mode. AbstractCPU also defines a set of rules that all brokers must abide by. Rules imposed on the brokers include details such as any execution within a broker object cannot block, and that operations must be performed in a timely manner[2]. These rules are similar to those imposed in Fluke, Aegis, and the Caching Kernel [53, 56].

Even though each broker may define a different communication model for its meta-space, the communication semantics are defined by AbstractCPU and remain consistent throughout Chameleon. The term *communication semantics* refers to the relationships among the various objects, and how messages are actually delivered among these objects. Communication semantics are defined with the exported interface and the broker interface. The term *communication model* refers to details such as blocking vs. non-blocking sends, and whether timing information (for a real-time object) is attached to a message. This design constraint is necessary for migration and for the usage of AbstractCPU's exported interface. It is also necessary for brokers to co-exist and work together, and for broker interfaces to remain consistent with AbstractCPU. This design constraint has not adversely impacted application-specific implementations of policies.

Exactly one *locus of execution* is associated with AbstractCPU. This approach remains consistent with the active object model defined in Chapter 2. AbstractCPU does not follow a multithreaded micro-kernel structure. This locus of execution is unique in that it executes in the CPU supervisor mode and is *not* a schedulable entity, just as a traditional micro-kernel isn't schedulable. Its execution begins in reaction to a software

---

[2] The execution time of any broker will determine the system response time. Computationally expensive operations that may introduce unacceptable response times, as determined by the applications and users of the system, should be transferred to a schedulable, preemptable active object.

or hardware event (exception or interrupt) and so is invoked by the CPU itself. It executes with interrupts disabled.

There is a variety of events that AbstractCPU must contend with: hardware interrupts, software exceptions, and general messages needed for communication among active objects and brokers. Hardware interrupts and software exceptions are abstracted as message objects so that the model defined by AbstractCPU is unified: its task is to deal with messages, regardless of the origin or purpose. This unification simplifies both the broker interface and AbstractCPU's implementation. A positive side effect is that the model for broker implementations is easier to follow when compared to normal kernel models, where these events are treated differently. This also makes quick prototyping and testing of new broker implementations possible.

AbstractCPU only contains functionality that is shared among brokers and does not include any implementation for the broker interface. It must dynamically bind itself through the class hierarchy to a broker for these method implementations. When AbstractCPU is bound to an instantiation of a broker, the combined object is guaranteed complete because of programming language rules: a broker cannot be instantiated if it is missing any method implementations, and AbstractCPU is only missing implementations for methods that are provided by brokers. AbstractCPU must bind itself to the broker whose object introduced a message to the system. The other task for AbstractCPU is to coordinate among all brokers in the system. AbstractCPU is defined as one class because the broker coordinator portion interfaces directly with brokers, and these may in turn need services provided by the hardware abstraction. Separating these concepts into different classes would change the inheritance rules applied to AbstractCPU and the brokers. This leads to a description of the two conceptual parts of AbstractCPU: *hardware abstraction* and the *broker coordinator*.

```
void AbstractCPU::genericInterruptHandler(int whichInterrupt) {
        push_currentLocus_user_registers_onto_stack
        load_AbstractCPU_registers_and_stack
        if (whichInterrupt == GENERIC_SOFTWARE_INTERRUPT) {
                Message *m := getMessage(currentLocus);
                if (m->GetRTTI()->IsValid() = FALSE) {
                        storeError(currentLocus);
                } else {
                        m->sender := currentLocus;
                        brokerCoordinator(m);
                }
        } else {
                if (handlerInstalled(which_interrupt) = TRUE) {
                        Message *m := invokeHandler(whichInterrupt);
                        If (m->RTTI()->IsValid() = TRUE)
                                brokerCoordinator(m);
                } else {
                        Message *m := message_representing_interrupt
                        m->sender := m->receiver := currentLocus;
                        brokerCoordinator(m);
                }
        }
        store_AbstractCPU_registers_and_stack
        pop_currentLocus_user_registers_from_stack
        return_from_interrupt
}
```

Figure 4.3: Pseudo-Code for AbstractCPU's Entry Code

4.3.2 Hardware Abstraction

This part of AbstractCPU interfaces directly with the hardware. Typically, this is the only code required to be rewritten when porting Chameleon to a different architecture. Any means by which the CPU can enter kernel/supervisor mode is routed through this code. Different actions may be taken depending on how the CPU entered kernel/supervisor mode:

- If a generic software exception occurs, then AbstractCPU knows that an active object has introduced a message to the system. In this case, the message is verified for basic correctness, i.e., it was initialized correctly so that its run-time type information (RTTI) is valid, the size is appropriate, etc. If the message is

valid, then the broker coordinator portion of AbstractCPU is invoked with that message. Otherwise, AbstractCPU returns execution to the active object with an appropriate error code.

- If an interrupt or exception occurs and a handler is installed for it (such as a timer or console), then the handler is invoked as a function call that is allowed to return a message. One example is the console object handler, which may introduce a message invoking its active object portion. In this case, the sender of the message will typically be different from the currently executing active object. If a message is returned and is valid, then the broker coordinator portion of AbstractCPU is invoked. Otherwise, AbstractCPU returns and the execution that was preempted for the interrupt continues.
- If no handler for the interrupt or exception has been registered with AbstractCPU, then an appropriate default message abstracting the event is created and passed to the broker coordinator portion of AbstractCPU on behalf of the currently executing active object. An example of this case is when a stack overflow causing a page fault occurs. The broker has the responsibility to perform any appropriate action for the event, and different events may trigger different actions.

This algorithm is outlined in Figure 4.3.

Here, an example, illustrated in Figure 4.4, is introduced to clarify the operation of AbstractCPU with respect to the broker interface. Assume active objects O_C and O_T are supported by the same meta-space M, which is abstracted by broker B_C. O_C constructs and initializes a suitable message M_{OC} to communicate with O_T. This message object's class ultimately inherits from the base Message class. Object O_C causes a generic software exception via a simple assembler instruction. This is how O_C calls AbstractCPU's exported interface that has one method. This instruction causes the CPU to enter supervisor mode where AbstractCPU first performs basic necessary tasks, such

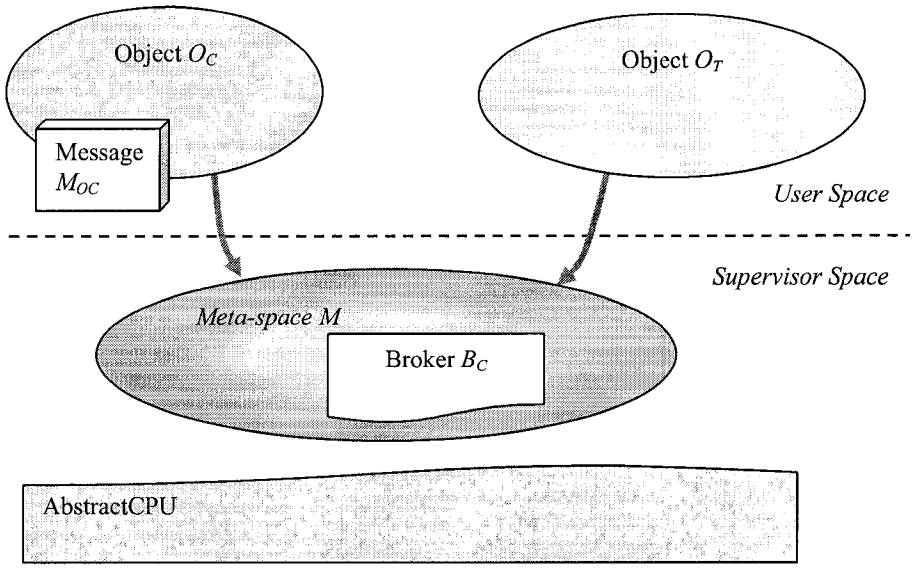


Figure 4.4: Component Layout for Communication Walk-Through

as saving registers and switching to a different execution stack. Then, AbstractCPU locates the message (the address of which is found in a specific register, but could have been pushed onto the stack) for examination. The message that O_C introduced, M_{OC}, is deemed initialized correctly so the operation can continue to the broker coordinator portion of AbstractCPU.

4.3.3 Broker Coordinator

The broker coordinator calls into the broker interface and is the second part of AbstractCPU's basic functionality. This portion of AbstractCPU is invoked only by the hardware abstraction portion of AbstractCPU, and is invoked for all valid messages introduced to the system. The basic algorithm for the broker coordinator is presented in Figure 4.5. Method calls into the broker interface are shown in boldface. Two variables that are specific to AbstractCPU, `currentLocus` and `primaryBroker`, are important in this algorithm. `currentLocus` is a handle to the locus of execution associated with

the currently executing active object. `primaryBroker` is a handle to the broker at the base of the meta-hierarchy. The actual broker interface is presented in Appendix B.

AbstractCPU must always be bound to exactly one broker. AbstractCPU first uses the current binding between itself and a broker to completely save the state of the currently executing object using the method called `saveContext`. AbstractCPU has already performed part of this task by entering supervisor mode (saving all CPU registers), but the broker may need to perform additional work, for example managing virtual memory page table entries or updating profiling information. Continuing with the example, AbstractCPU must first allow the broker of the currently executing object to finish saving any state information. O_C is the currently executing object so its broker, B_C, is invoked for this operation.

```
void AbstractCPU::brokerCoordinator(Message *m) {
        saveContext(currentLocus);
        Broker *b := m->sender->broker;
        Locus *l := NULL;
        do {
                bindToBroker(b);
                if ((m != NULL) AND (securityCheck(m) = FALSE)) {
                        l := currentLocus;
                        m := NULL;
                        bindToBroker(currentLocus->broker);
                } else {
                        l := resolveTarget(m);
                        if (l != NULL)
                                m := resolveMessage(l);
                        else {
                                m := NULL;
                                bindToBroker(primaryBroker);
                                b := resolveBroker();
                        }
                }
        } while (l = NULL);
        currentLocus := l;
        restoreContext(currentLocus, m);
}
```

Figure 4.5: Pseudo-Code for the Broker Coordinator

After the state of the currently active object is saved, AbstractCPU binds itself to the broker of the object introducing the message, which may not necessarily be the currently executing active object. This situation can occur when an interrupt takes place, where the interrupt handler is associated with a locus of execution that is different from the suspended active object. In our example, message M_{OC} originated from an active object and not an interrupt handler, and its sender field properly identifies O_C as the object introducing the message. O_C's broker, B_C, is identified by AbstractCPU so it re-binds to B_C. After this binding takes place, AbstractCPU treats broker B_C as a super-class.

The `securityCheck` method allows the broker to determine if the active object is allowed to introduce the message, and can perform further verification on the contents of the message. If it does not pass these tests, then the message fails with the appropriate error code. In this case, the execution continues at the last step of this process. `securityCheck`'s role is to act as a guard against illegitimate messages, similar to that found in the Spin operating system [12]. In our example, B_C's `securityCheck` method is invoked and the message is verified as acceptable.

Once the message passes the security check, it is passed to the broker via the `resolveTarget` method, which returns the locus of the active object to be executed next. AbstractCPU expects one of three possible values:

- A handle to the currently executing active object.
- A handle to another active object. In this case, the broker has somehow dealt with or queued the message and determined that a different active object that it manages should now execute, which may be the recipient of the message that was introduced.

- A NULL handle indicates that the broker has no object under its management that should execute at this time. This situation is described below.

If the `resolveTarget` method returns a handle for an active object to run, then the `resolveMessage` method is invoked. Here, the broker specifies the message (if any) to be delivered to the active object already selected to execute. It is expected that a broker will perform scheduling operations during the `resolveTarget` method call. In the example, B_C's scheduling algorithm returns O_T from the `resolveTarget` method call. Finally, B_C informs AbstractCPU that O_T is to receive message M_{OC} (originally introduced by O_C) when AbstractCPU calls the `resolveMessage` method.

The final step in this algorithm is a call to the `restoreContext` method, the inverse to `saveContext`. Here, the broker may place any message that is to be delivered in a location (such as in a register) that the target active object can expect. The current binding between AbstractCPU and the broker is retained for the next invocation of the broker coordinator. In our example, AbstractCPU has an object (O_T) and a message (M_{OC}) and finally calls B_C's `restoreContext`. The binding to B_C is retained for the next invocation into AbstractCPU. The broker coordinator portion of AbstractCPU has completed, so execution returns to the hardware abstraction portion, which switches to the appropriate (O_T) user execution stack, pops saved registers from the stack, and executes the assembly instruction "return-from-interrupt" so that O_T can begin execution.

The algorithm is straightforward except for the case in which AbstractCPU cannot determine which active object should execute next, i.e. when a `resolveTarget` returns a NULL handle. In this case, AbstractCPU unbinds itself from the current broker and binds itself to the primary broker (discussed later) to call the `resolveBroker` method to determine what broker to bind to next. Brokers may call other brokers' `resolveBroker`

methods to determine a meta-space containing an active object that is ready to execute. Once an appropriate broker is identified, AbstractCPU binds itself to that broker and calls its `resolveTarget` method with a NULL message handle, signifying that no message was introduced, but that the broker must now select an active object that it manages to execute. This technique is also used for preemptive scheduling. The 'idle' active object, managed by the primary broker, will execute if there is no other active object in the system that can execute. The primary broker uses the idle active object to prevent the loop in broker coordinator algorithm, presented in Figure 4.5, from proceeding beyond a second pass.

In the process of describing these operations, the basic broker interface has also been presented. This interface is exploited by AbstractCPU and by brokers to create Chameleon's flexibility. There are many important points to note:

- Preemptive scheduling is intrinsically supported by Chameleon. The clock interrupt handler always forces the invocation of the primary broker's `resolveBroker` method, which will elect another broker and meta-space to begin execution.
- Every broker has complete control over which of the active objects that it supports will execute next. Thus, AbstractCPU introduces no policy decisions affecting active object scheduling.
- Rather than aiming for an expressive interface or set of messages to handle all cases known and expected in the future, a suitably abstract broker interface is defined to handle all cases with respect to broker implementations *and* broker dictionaries. This fact means that the system can continue to evolve by simply introducing new brokers, and with very minor, if any modifications to AbstractCPU and the rest of the system. Issues with the adaptability of an expressive interface have been completely circumvented because AbstractCPU

and the broker interface are separate from and are unconcerned about particular messages that comprise a specific broker's dictionary.

- The conventional approach is that an event-driven system works within the confines of a pre-determined set of events. In our approach, however, broker objects remain event-driven (they act upon the receipt of a message) but instead are expected to define the set of acceptable events (messages). This set is allowed to change and will include but is expressly not limited to some messages that are pre-defined by the system.
- Brokers with varying policies can exist simultaneously and relatively independently. Policies are independent but meta-spaces must arbitrate for and share the finite base of resources.
- AbstractCPU follows the active object model in that it is not re-entrant, which contrasts with many current kernel designs that are multi-threaded. This approach is a simpler model for the systems programmer to follow, which should inevitably result in fewer bugs. However, it also means that a particularly inefficient or poorly designed broker may impede overall system performance. The designer of a broker must carefully weigh these design considerations and possibly offload expensive operations to active objects for concurrency.

4.3.4 Dynamic Class Binding

AbstractCPU differs from Apertos' MetaCore. Where MetaCore, or any micro-kernel is stand-alone, AbstractCPU is not. AbstractCPU is implemented as a virtual (or abstract, or incomplete) class and cannot be instantiated as an object on its own due to programming language rules: AbstractCPU lacks implementations for methods in the broker interface, and so cannot be created on its own, since any calls to undefined methods will result in a failed execution. AbstractCPU must dynamically bind itself to a specialized sub-class, i.e. a broker, via the broker interface, an operation referred to here

as *dynamic class binding*. This operation takes place in the `bindToBroker` method used in Figure 4.5. Once this binding has taken place, normal inheritance (in the broker class hierarchy) rules apply and is straightforward because AbstractCPU does not provide any implementations for methods in the broker interface and does not define any other virtual functions. This dynamic class binding technique is relatively cheap (costs are given in Chapter 7), and is achieved by using object-oriented constructs such as inheritance; this concept is beyond the bounds of traditional structured programming, unique in operating system design, and is alien to the micro-kernel model.

Typically, there is type information associated with each object, which is used to identify its class. This type information is used to invoke the correct implementation of a method, where the super-class's method may be replaced by a sub-class (e.g. using the `virtual` keyword in C++). AbstractCPU applies the type information for the broker of the requesting object to itself. Once this type information is applied, calls via the broker hierarchy are invoked in a manner consistent with any other class hierarchy. The cost of this binding is constant is independent of the number of brokers instantiated in the system.

4.4 Brokers and Hierarchies

Brokers work together within the structure imposed by AbstractCPU to facilitate a highly reconfigurable and extensible system. The functions of a broker are defined as follows:

- to act as a communication medium between an object and its meta-space (for communication in either direction);
- to identify and manage the collection of service objects comprising a supported active object's meta-space;

- manage, or delegate management of, any supported active object's external state information (such as open file handles) so that, for example, appropriate clean-up actions can take place when an active object terminates;
- to interpret and manage the *locus of execution* for any active object that it supports;
- to facilitate migration, either between machines or on the same machine and to ensure that object migration can be performed safely (for example, avoiding migration during some execution with soft real-time guarantees); and
- to handle any programming language specific dependencies, such as the storage format of a text string.

The basic implementation of a *locus of execution* object is determined by the hardware, and can only save or restore the CPU state going to/from kernel/supervisor mode. It also contains a handle to the broker managing it. It is therefore also used as an active object identifier (OID), which is analogous to a traditional system's process identifier (PID). It is intended to be sub-classed by a broker to introduce notions such as scheduling priority, message queues, profiling information, etc. that a broker may need to implement its policies. Therefore, a locus of execution class hierarchy will be roughly parallel to the broker class hierarchy. That is, often, any broker will typically have its own locus class definition that inherits from the broker's superclass's locus definition. Appendix A shows the actual class hierarchies in the implementation of Chameleon, and shows the parallel nature of the broker and locus class hierarchies.

Despite the dynamic binding between AbstractCPU and any broker, brokers must still be arranged in a hierarchical manner, for resource management, that will be different from the broker class hierarchy. This hierarchy defines the run-time dependencies among the brokers, which is dynamic and by definition is the meta-hierarchy.

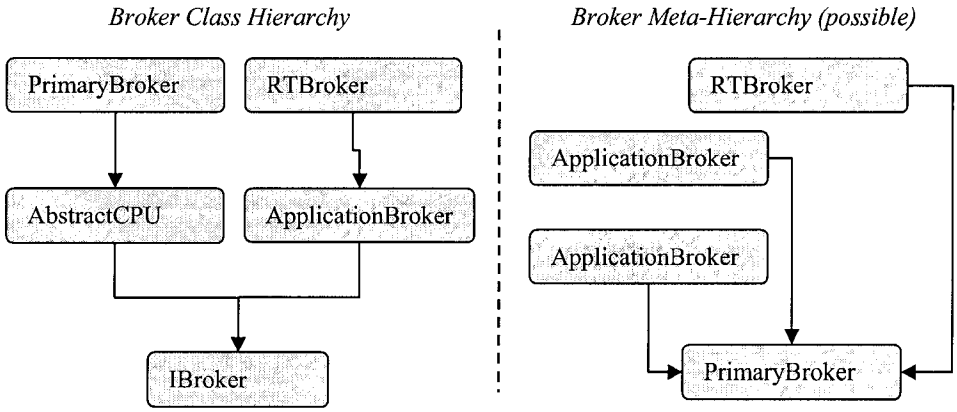


Figure 4.6 Broker Hierarchies

At the base of the meta-hierarchy is *PrimaryBroker*. It is not dependent on any other broker in the system for resources such as CPU clock cycles. All other brokers, either directly or indirectly, must negotiate for resources that are ultimately delegated by PrimaryBroker. Figure 4.6 shows the actual broker class hierarchy alongside a possible run-time meta-hierarchy. The meta-hierarchy does not show AbstractCPU because all brokers use it. Two ApplicationBrokers and one RTBroker depend on PrimaryBroker for basic services at run-time, even though they do not inherit from PrimaryBroker for implementations. This class hierarchy is present in the Chameleon implementation, and is used in the following diagrams.

There is exactly one PrimaryBroker object instantiated in a Chameleon system. At system start-up, PrimaryBroker and AbstractCPU are instantiated together and provide a minimal support system for all other services. PrimaryBroker is designed to be dynamically replaceable, just as every other object in the system is, with the exception of AbstractCPU. A conceptual layout of Chameleon brokers and AbstractCPU was given in Figure 4.2. Figure 4.7 illustrates the run-time relationship after AbstractCPU dynamically binds itself to PrimaryBroker.

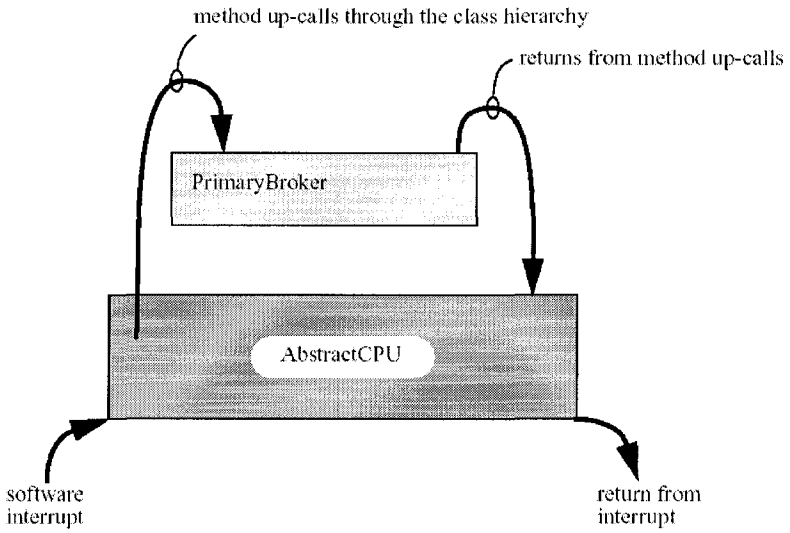


Figure 4.7 AbstractCPU and PrimaryBroker

Other brokers may exist in the system during run-time. These brokers are allowed to abstract different meta-spaces with different policies. Figures 4.7 and 4.8 illustrate how AbstractCPU binds itself to other brokers. Notice that these dynamic class bindings are equivalent from AbstractCPU's perspective. Method up-calls from AbstractCPU only occur through the broker interface, via the algorithm in Figure 4.5. Method down-calls to super-classes as shown in Figure 4.9 are defined by the class hierarchy at compile-time, are allowed by the programming language, and are not over-ridden by AbstractCPU's dynamic class binding.

Brokers must directly communicate among each other to handle such tasks as inter-meta-space communication, migration, name look-ups, and the creation and deletion of meta-spaces (upon creation and destruction of brokers, to manage the meta-hierarchy). Another method in the broker interface, called `brokerMessage` provides this functionality. This method takes as parameters a handle to the requesting broker and a handle to an object whose class inherits from *Message*, and returns a result code. Any return values are stored in the message object.

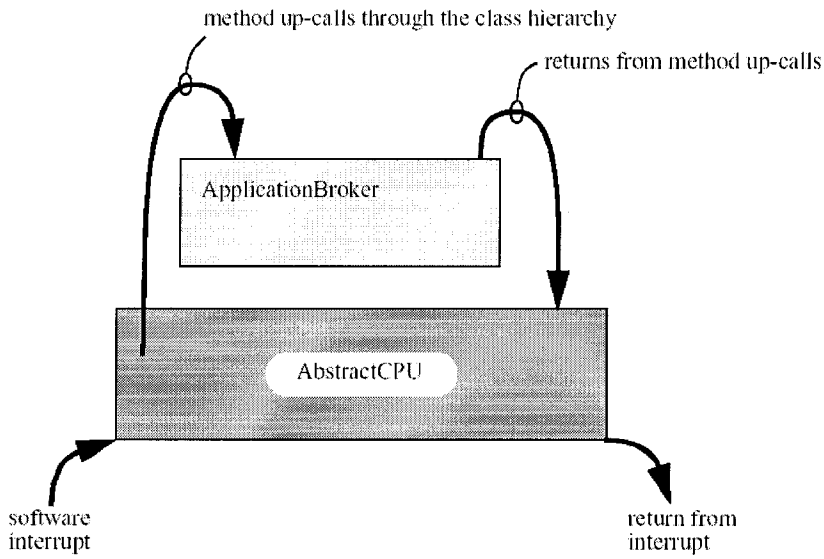


Figure 4.8 AbstractCPU and ApplicationBroker

Every active object must be associated with exactly one broker at all times, and thus AbstractCPU's binding is always deterministic. This rule means that the management of active objects (particularly of interest are service objects) cannot be shared by meta-spaces. However, this rule simplifies a lot of potential problems relating to resource management and does not impose any undue performance issues when the `brokerMessage` method is used properly.

4.4.1 Attained Flexibility

The flexibility achieved by the AbstractCPU/broker inheritance model is quite powerful. PrimaryBroker, for example, should implement policies in a very efficient manner, since all other meta-spaces are functionally dependent on it. It can make some assumptions about the objects it manages in order to improve efficiency, since these objects are a well-defined subset of the objects in the system. It is analogous to the smallest, most efficient micro-kernels for embedded systems.

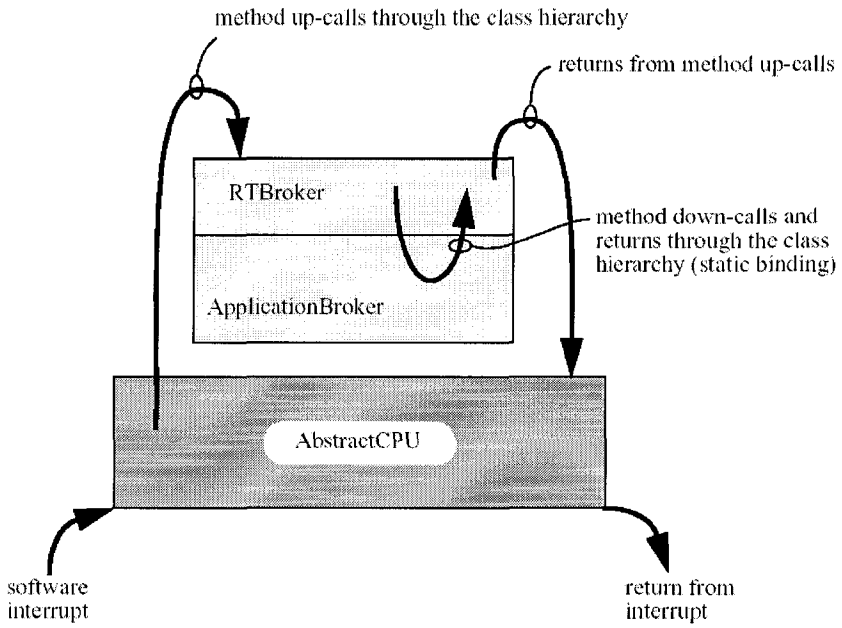


Figure 4.9 AbstractCPU and RTBroker

Application objects are expected to execute in a meta-space defined by *ApplicationBroker*. The broker interface method implementations for ApplicationBroker cannot be as efficient as PrimaryBroker because more verifications have to be performed on the messages that it receives. Still, it may be possible for ApplicationBroker to enact many of these policies in supervisor mode without adversely impacting the overall system performance.

Active objects requiring persistence could be managed by *PersistentBroker*. It is a good example of a broker that will need to defer some calculations to a service object. If it receives a message destined for an object that is currently swapped out of memory, then the message should be examined and queued. Actions to swap the target object back into memory are very time-intensive, and should be deferred to a schedulable, preemptable entity.

These examples illustrate that AbstractCPU treats all communications among objects in exactly the same manner. In the first example, PrimaryBroker quickly passes a message to its target where in the second example, verifications were performed by ApplicationBroker. In the final example, at least one service object was invoked to correctly deliver the message to the target object. In this respect, the Chameleon model achieves an important goal: to efficiently and consistently support substantially different policies on the same architecture. This goal is vital to providing optimized support to both multimedia and database applications simultaneously.

In addition to this flexibility in design, the Chameleon model supports easy upgrading and replacement of service and broker objects. *AbstractCPU* performs the dynamic class binding, which means that the requesting active object does not need to be aware of the broker that it is associated with it. The requesting object merely needs to introduce a message to the system (via a generic software exception, causing *AbstractCPU* to begin execution). The supporting system does the rest of the work needed to deliver the message. Replacing one broker with another (to upgrade or to change the policies) can be performed because dependencies are well defined and active objects do not have direct connections to their brokers. Brokers are responsible for determining when replacement is safe to perform.

4.4.2 Layers of Abstraction

Brokers provide an important layer of abstraction between application objects and service objects. Service objects are useful because they are allowed to block for specific events, whereas brokers are not. This layer of abstraction also limits the assumptions that the application object needs to make about a service object. The broker object can act as a "translator" from its API that the application object uses for communication, to the

interface implemented by the service object[3]. This extra layer of abstraction increases the potential number of service objects available to the application objects. However, the broker must also understand the service object's interface, which is reasonable since the broker has control over the content of its meta-space. Replacing a service object can then be performed in a fashion that is as safe as replacing a broker - so long as the broker prevents the requesting active object from obtaining a handle to its service object, and communications to service objects are only handled by the broker.

In practice, the number of meta-layers is expected to be three: the low-level system layer, an application-specific support layer, and the application (base-level) layer. This configuration is illustrated in Figure 4.2. However, there is no reason to impose this limitation onto the design. One obvious extension would be the addition of an application-specific layer to provide services to a highly concurrent and frequently communicating layer of background active objects, similar to application thread libraries. An application-specific broker may also support an interrupt handler for a dedicated device such as a scanner. There is no performance penalty incurred from the broker hierarchy when an interrupt handler is supported by a broker other than the PrimaryBroker.

4.5 Security and Trust

Although the security of Chameleon is not a focus of this work, it is an issue that is universal to any extensible system. Sandboxing and protection guaranteed by the programming language are two solutions. Paramecium, another research operating system [63], represents a different solution that is a generalization of the approach taken

---

[3] This comment refers to normal operation. If an object chooses to send a message directed at a particular service object rather than using the API provided by its broker, then the target service object must support the proper interface.

in Spin, using well-defined interfaces and object certification. Simply preventing application code from being inserted into the operating system is another solution. With respect to the goal of extensibility rather than just adaptability, this work aims for a compromise: the bulk of work is placed outside what can be considered the kernel, and the small set of objects (brokers) executing in supervisor mode are designed so that they seldom need replacing or extensions. When changes are necessary, the duties and interfaces of these brokers are strictly defined.

The `securityCheck` method provides all brokers with a mechanism to verify the validity and security of any message that they receive. However, the Chameleon model assumes that all brokers themselves are trustworthy and may be considered secure. Since an active object must communicate with its broker via AbstractCPU, the system is only as secure as the broker. AbstractCPU is considered safe from malicious or malformed messages because it never examines the contents of a message, other than to verify that RTTI has been correctly initialized.

Brokers are not unique in executing in supervisor mode. Every interrupt handler and device driver will require the ability to execute privileged instructions, but must still be associated with a broker. It is responsibility of that broker to verify the authenticity and correct behaviour of the privileged object.

Chameleon is not inherently protected from an errant broker or interrupt handler (e.g., one that is under development). In this situation, brokers and interrupt handlers executing in supervisor mode are expected to provide their own detection and recovery mechanisms, such as sandboxing.

Moreover, brokers must cooperate amongst each other to maintain an acceptable level of performance and resource sharing. Thus, broker replacements should be limited to a small group of trusted agencies rather than any application provider. This approach

is considered a major improvement over other extensible systems like Spin where any application may inject code into a kernel.

4.6 Real-time Guarantees

Timing costs in AbstractCPU are deterministic. The cost of binding to a broker is constant, and only has to be repeated in specific situations where the current broker has no object to execute. However, costs of method invocations via the broker interface may vary with each broker. Therefore, any of Chameleon's real-time guarantees assume the same type of trust as the security guarantees discussed above. Still, with respect to guaranteeing the soft real-time constraints of multimedia systems, this aspect of Chameleon is superior to the Spin and Vino designs where any number of event handlers may be installed on a particular event. In Spin, the cost associated with an event will change every time a handler is installed or removed from the system because each handler's guard and potentially many handlers will be invoked. The variable overhead in the dispatching mechanism also has to be considered, as one would expect its performance to decay linearly with the introduction of each spindle. Even though these models can limit the number of handlers for each kernel event, a variety of kernel events may be triggered for a given event.

Given the trust applied to brokers in the system, Chameleon can statistically provide a level of accuracy to soft real-time guarantees. PrimaryBroker is invoked every time preemptive scheduling occurs and will elect any single broker for execution. This election scheme can be designed to incur a relatively constant cost regardless of the number of brokers instantiated in the system. Therefore, any real-time broker can provide the same level of guarantee that PrimaryBroker can provide to it.

Chameleon can also provide hard real-time guarantees when the frequency of asynchronous events is known and all extensions provide detailed data regarding

execution times and resource requirements. Hard real-time systems [64, 65] are used in situations where if a task does not complete within a time limit, then results may be catastrophic. Hence, these systems are typically closed systems where all combinations of events are known and proof of correctness of the system can be guaranteed analytically, not statistically.

4.7 Broker Conclusions

The design of the broker hierarchies is influenced by previous research, much of which is outlined in Chapter 3. As such, it is useful to contrast this design with existing work.

The closest analogy to the Chameleon broker system at run-time is a collection of micro-kernels sharing the same hardware abstraction code and cooperating to share the same hardware resources. In Chameleon, each of these micro-kernels (i.e., brokers) has its own operating system layer (meta-space) and applications (base-level active objects). Communication can take place among these mini operating systems. One main micro-kernel (PrimaryBroker) acts as a final, global manager of resources. The partitioning of functions among service objects is at the discretion of the designer of that meta-space (or mini operating system or environment) and is not imposed by the Chameleon design. This aspect of partitioning is important because new operating system technologies and hardware platforms will continue to redefine the "correct" set of functions that are to be included in a support layer for an application. Where other systems need to be redesigned, Chameleon will need to have a new meta-space implemented, and the new meta-space can execute along-side old meta-spaces.

The broker hierarchy architecture of Chameleon is essentially the inverse of event-centric architectures found in Spin or Vino. In these systems, the kernel notifies all extensions to an event when that event occurs. In Chameleon, the architecture is not

concerned with a particular event or message; it is only concerned with electing a single broker to handle the message. The broker's task is to then deliver the message to target active object. Thus, even when comparing perfectly behaving systems, Chameleon should achieve better performance with respect to handling a particular event. On the other hand, systems such as Spin and Vino may outperform Chameleon when sending a "broadcast" message destined for many objects. The broadcast dispatch mechanism is an important aspect of Spin and Vino, where the Chameleon model is optimized for delivery of messages to specific targets.

4.7.1 Brokers versus Reflectors

From another perspective, brokers appear to be similar to Apertos reflectors [15, 39]. Their duties to manage a meta-space are comparable, but there are two fundamental differences in these approaches:

- Apertos reflectors are implemented as active objects, and therefore must be explicitly scheduled. This requires the invocation of the meta-space supporting the reflector (the meta-meta-space) to ensure consistency of the system model. Consequently, every meta-space ultimately supporting an active object is invoked during any form of communication. This fundamental design decision has severely impacted communication performance due to excessive context switches [16]. Chameleon takes a contrasting viewpoint where brokers are recognized as providing a unique service in the system and therefore are not simply active objects. All brokers execute in CPU supervisor/kernel mode and are not schedulable entities. Therefore, only the broker supporting an active object is necessarily invoked during communication. Any other meta-space invocation will be due to the broker's policy and not the system's design. In our previous example, only broker B_C had to be invoked for communication between O_C and O_T. In Apertos the reflector supporting O_C and O_T would have to be invoked, but

all meta-spaces ultimately supporting that reflector (usually two) would also have to be invoked.

- Because Apertos reflectors are simply active objects, another meta-object (called MetaCore) is necessary to define another type of *meta* for communication between the different meta-layers. MetaCore is essentially a micro-kernel and executes in CPU supervisor/kernel mode. This second form of *meta* exposes a confusing inconsistency between Apertos' theoretical model and its implementation. This second form of *meta* is not necessary or present in Chameleon. In Chameleon, when an active object needs to access its meta-space, it does so via its broker.

4.7.2 Summary

This chapter has defined and explained the design of AbstractCPU and the broker hierarchies in the Chameleon model. In addition, the flexibility achieved has been outlined and the work has been contrasted against related research. The Chameleon broker architecture represents a significant move towards a safe, extensible operating system and is an evolutionary next step from conventional monolithic kernels and micro-kernels.

Chapter 5

Memory and CPU Management

In Chameleon, brokers are the crux of a system where all constructs are objects. Functionalities normally found in applications and server processes are implemented as active objects so that they are schedulable entities that the operating system is expected to serve and manage. Active and passive objects may be loaded into memory or stored on some permanent medium such as a hard drive. This chapter describes low-level management of memory and CPU resources, tailored to support these constructs, to maintain consistency in design and to provide mechanisms to maximize the potential of the broker hierarchies.

5.1 Memory Management

Performance bottlenecks in traditional systems have been identified as context switching, crossing memory protection boundaries, and inter-process communication (IPC) [6, 57, 66, 51]. These bottlenecks pose a concern to object-oriented systems due to their fine-grained nature: numerous communications among small objects [15, 38]. Chameleon exploits strongly defined interfaces and dependency graphs, but at the same time requires much explicit communication to take place among active objects, which are protected entities. Therefore, communications across protection boundaries have to be optimized. In turn, the memory model, relatively unexplored in object-oriented operating systems, has a significant impact on communication performance. This chapter discusses the memory model and file system for the Chameleon object-oriented design and presents reasons for the decisions made. Primitives used to implement the memory model are also defined. Communication is covered in the following chapter.

5.1.1 Single Address Space

SASOS (or Single Address Space Operating System) [23] designs separate the concept of protection domains from the concept of memory addressing, contrary to the vast majority of systems, including UNIX. A *protection domain* is defined as the collection of memory addresses that a given object can access. In a SAS design, everything works in the same address space, and any virtual address in any protection domain is guaranteed to be unique in the system. Different pages of memory within a shared global virtual address space can be mapped as accessible to specific objects. A SAS design is chosen in Chameleon and adapted to retain protection while increasing communication performance.

Figure 5.1 illustrates how many protection domains can exist in one common virtual address space. All addresses are unique. Two protection domains can share one

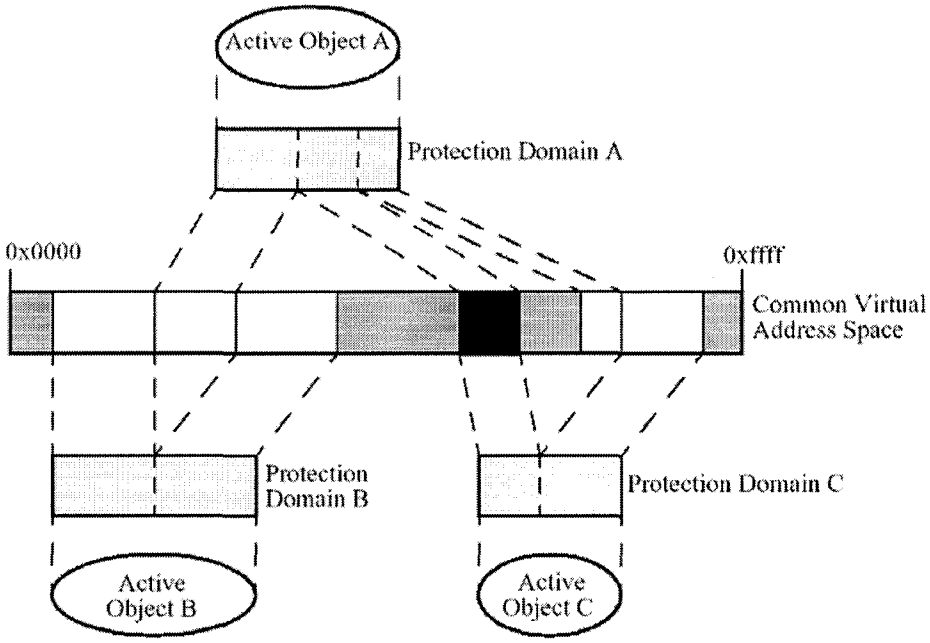


Figure 5.1 A Single Address Space with Multiple Protection Domains

segment of memory, shown as the solid area. A protection domain does not have to be contiguous in memory. The greyed areas in the common virtual address space illustrate memory address ranges that are not associated with any protection domain (so represent unallocated memory). A meta-space containing active objects A, B, C (not shown) will typically be expected to be a superset of the combination of their protection domains.

A number of SASOS prototypes have been implemented to date [30, 67, 68], and previous studies have shown that a SAS design will simplify overall system design and often improve communication performance. For example, a kernel does not have to deal with address space domains when examining a user application's address space where systems such as UNIX do [67, 68, 69]. Private data can also be easily converted to shared, and cache aliasing is eliminated [70]. Finally, SASOS presents many advantages in hardware virtual-physical translations [71].

Opal [67, 70, 71] is a project based out of the University of Washington, Seattle whose primary focus is on sharing information and exploring asymmetric trust relationships: where a client may trust a server with shared data regions, but the server is not required to trust its clients. The file system is eliminated, and disk storage is used only as backing store for virtual memory. Thus, objects can stay at the same virtual address for the duration of their lifetimes. One drawback to Opal is that it assumes that objects are reentrant and thread safe, and the programmer has to explicitly call locking primitives to deal with synchronization [23, 67].

Mungi [68, 72] also eliminates the file system, favoring mapping all objects into a distributed address space for all time. Mungi is implemented using a modified L4 kernel [56], and password capabilities are implemented to maximize overall performance. Mungi defines *protection domain extensions* (PDX) whose purpose is to define a new

protection domain for each invocation to a protected object. The new protection domain is the union of:

- the target object's protection domain; and
- the caller's protection domain, or a protection domain explicitly defined and provided by the caller (to provide finer grained protection).

A separate task is created to use the new protection domain, to execute the called, passive, protected object. Mungi also supports user-level page fault handlers [72] which appear to be analogous to Spin's spindles [45], although implementation may be entirely different.

Grasshopper [30] is not a true SASOS, but presents a generalized memory model that can emulate both a traditional model found in UNIX, and a SASOS model. Many advantages are cited, however, this hybrid memory model can lead to ambiguous pointers and confusion from the programmer's perspective [71]. Grasshopper also defines a recursive container mapping system which can be created independent of any thread of control, and is used to construct collections of statically linked libraries which can be instantiated before application run-time to cut down on the costs associated with application start-up. The recursive container mapping also provides an interesting solution to handle shared, protected entities.

A primary reason for this interest in SASOS designs is that new processor families are supporting very wide address spaces. Studies have shown that address spaces of 2^{64} bytes in size will be large enough for beyond the foreseeable future [73]. However, most SASOS designs have concentrated on the advantages of easier sharing of information across protection domains and eliminating the file system. These approaches are based on a persistent address space where once an object is instantiated, it will remain at a given address range regardless of what machine it has migrated to, or even if it has been stored to permanent media.

In Chameleon, the file system is retained, and the emphasis is high-speed, efficient communication even in light of exceedingly large and frequent messages, as are found in multimedia applications such as real-time video transfer. Sharing is not precluded. And, if an object is not presently needed, it may be stored to disk, destroyed, and its address range(s) returned to the free pool. After an object is migrated or restored from permanent media, it may be at a different location in memory. Shared memory, which may, for example, contain shared libraries, becomes an issue when an object is stored or migrated. These situations are specific to the nature of the object and appropriate techniques are left to the designer(s) of the objects sharing memory. No special techniques are needed for communication with traditional systems, other than sharing the same protocols.

5.1.2 SAS and Chameleon

There is only one protection domain for each Chameleon active object. Passive objects may also be instantiated within that protection domain. A protection domain is either:

- unique to that active object and associated with, and created and destroyed with its locus of execution, or
- defined by, created, and destroyed with that active object's meta-space, much like a thread works within the confines of its application's protection domain.

Any memory management policies are defined by the broker. It is expected that these operations will take place in the calls `saveContext` and `restoreContext` in the broker interface. This rule aids in the creation of extremely lightweight active objects sharing a common protection domain when necessary (where little or no work is required in the `saveContext` and `restoreContext` methods), and more heavily protected entities when policy dictates so. Fluke presents a similar hierarchical memory

management scheme [1] but is implemented in a substantially different, less consistent manner.

In Chameleon, protection domains are allowed to overlap. If memory cannot be shared among active objects, then run-time type information (RTTI) must be duplicated in every protection domain and either performance will suffer or a multiple-update problem will exist when new classes are introduced to the system. Therefore, memory is shared across protection domains, and the same memory addresses are to be used so that the objects representing classes for RTTI can remain valid in every protection domain. Protection domains may also contain memory which is marked read-only for (e.g.) executable code.

5.1.3 Memory Management Primitives

All memory actions are performed using the primitives given in Table 5.1. This list of necessary memory primitives is very similar to that provided in [56], though their application and semantics are modified for SAS. All of these primitives work with *virtual memory segments*, defined as one or more virtual memory pages. This definition of virtual memory segments is consistent with that of Opal [67]. Virtual memory segments have contiguous address ranges, but can be combined with or split into other memory segments.

All of these primitives are allowed to fail, for various reasons (e.g. unavailability of virtual memory for an `allocate` request, or a request to transfer the ownership of memory pages by an object that does not presently own them).

Meta-spaces will want to extend the functionality of these primitives. For example, the policy of the `elect` primitive may impact an application's performance, so it should be specific to the type of application. More familiar memory management

routines, such as `malloc`, `free`, etc., are also provided by meta-spaces and application libraries are built atop these primitives.

5.1.3.1 Dynamic Memory Allocation

An objects has the ability to request for memory from its meta-space by using the `allocate` primitive. A memory service object, in turn, has the ability to donate, or re-map some of its memory pages to objects that it directly supports by making a `transfer` request.

When the meta-space does not have enough memory to transfer to the application, it may perform its own `allocate` request to its meta-space. Such `allocate` requests may recurse to the base of the meta-hierarchy where a physical page of memory may be elected and possibly assigned a new, unique virtual address. Note that this action may in fact generate an `elect` request back up the meta-hierarchy to determine which physical page will be swapped and assigned the new virtual address.

When an object wishes to return memory to the system pool, it does so using the `transfer` primitive. The meta-space then has the option of transferring said memory to its meta-space (meta-meta to the original object) or to return the memory to the system pool. Correspondingly, the meta-space has the option of pre-allocating blocks of memory for its objects (which may be required in real-time applications). That is., a meta-space

Table 5.1 Memory Primitives

Method Name	**Method Action**
`allocate`	request for new memory
`transfer`	transfers the ownership of memory pages from one active object to another on the same host (includes allowing sharing of said memory pages)
`elect`	request for a segment that can be swapped to disk space

has the option of 'caching' memory blocks for efficient application-level memory allocations, as is done by some application software development libraries.

The `transfer` primitive does not rule out sharing, so a meta-space may retain access to a memory block even after ownership is passed to another object. Thus, any virtual memory page can be arbitrarily marked as private or shared (with either read or read/write privileges) among an arbitrary collection of objects. Combined with a single virtual address space common to all objects, this memory primitive, for example, significantly simplifies the implementation of a debugger. Sharing pages of memory marked as read-only also avoids unnecessary duplication of object code[4].

The `elect` primitive is used to determine an appropriate page of memory to be swapped. However, because AbstractCPU is not re-entrant and cannot block to wait for a page to be swapped back into memory, there are certain pages of memory, such as those containing loci and those containing execution stacks, that should never be swapped out. This primitive must be aware of these constraints while selecting memory pages.

The recursive nature of these primitives distributes responsibilities among a collection of cooperating memory service objects (possibly implementing different policies) in a manner that is unique to very few operating systems [1]. This distribution forms a hierarchy of memory service objects and will let different memory management policies (such as LRU vs. MRU) to co-exist. A single address space simplifies the implementation of these primitives immensely. Defining a closed set of memory management primitives, given in Table 5.1, is needed to support the dynamic and recursive nature of Chameleon's structuring concept. Virtual addresses do not have to be translated among active objects (because all objects work within the same address space),

---

[4] Self-modifying code is not considered in this work.

and page table entries only have to be added and removed from various objects' page tables.

5.1.3.2 Memory Primitives and Communication

The `transfer` primitive is used in response to allocation requests. It is also used in communication to transfer access rights of a message object (and any related object parameters passed by reference) into the target active object's protection domain. Since a virtual memory address is guaranteed to be unique among all active objects, explicit copies are avoided and pointer manipulation in composite objects does not have to be performed. Figure 5.2 illustrates how a message would be delivered from the requesting object (active object 1) to the target object (active object 2) where the objects are located on the same machine. Active object 1 has already constructed the message object.

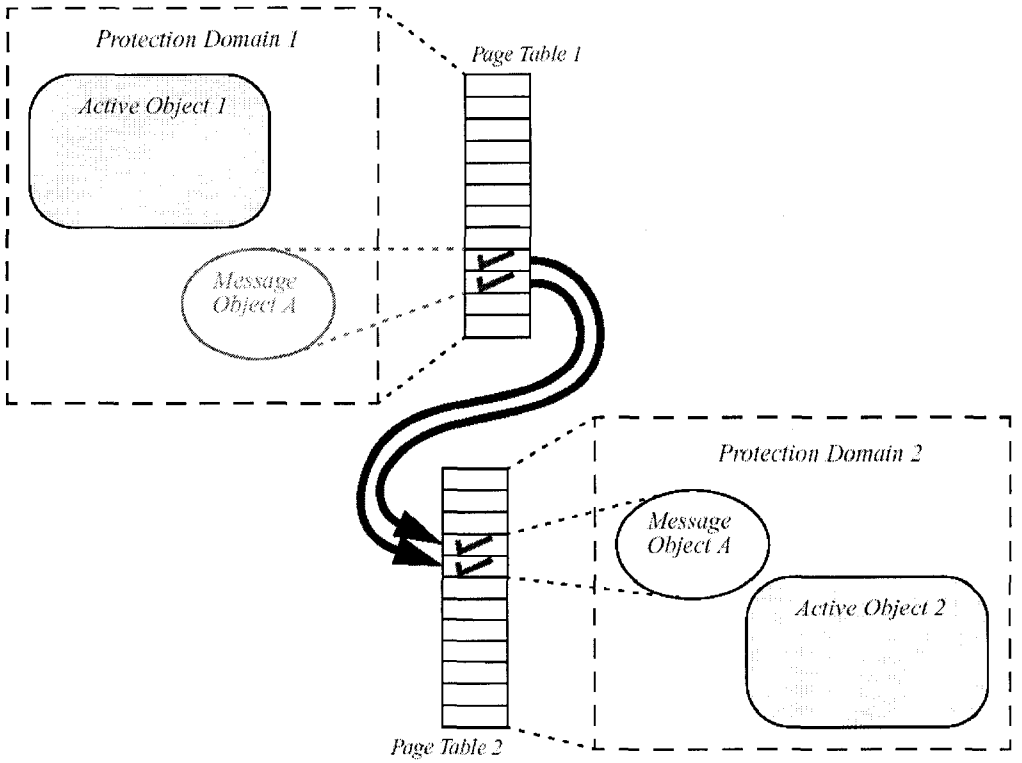


Figure 5.2* transfer *Mapping a Message From One Object to Another

During the call to `transfer`, its page table entries that locate the message object are copied into available entries in the target object's page table. Any read-only memory pages containing the object's executable code are also transferred. The message object itself is completely unaffected and only the target protection domain is modified. Upon the call's completion, the target object's protection domain (protection domain 2) is modified again to revoke access to the pages holding the message object, and the requesting object can continue.

Clearly, this operation is far more efficient than explicit message copying as is typically performed in systems like UNIX. In fact, a traditional message copy could incur a greater overhead in Chameleon because a broker must complete its operations in a timely manner to prevent adversely affecting total system operation. This means that the copy would have to be off-loaded to a special active object that can peer into both the requesting and target objects' protection domains, or perform the operation over a sequence of calls through the broker interface.

During asynchronous calls, the message object may have to be mapped out of the requesting object's protection domain. Such specifics are left to the meta-space implementing the message primitives that use `transfer`. What is of consequence here is not the particular semantics or model for a message send, but of the underlying tools provided by the memory subsystem that are used to send messages, and to maximize communication performance.

Communication based on these techniques - just transferring ownership of memory pages - has significant advantages in multimedia applications where large blocks of data are the norm. [60] implements an efficient multimedia device driver in UNIX by modifying the UNIX system to avoid massive copies between virtual address spaces by transferring ownership of physical memory pages. [30] outlines how some traditional micro-kernels use techniques such as memory-mapped files to avoid large copies of data.

These operations are somewhat analogous to the Chameleon `transfer` method, but this system provides such functionality intrinsically and consistently for all message passing.

5.1.4 Security and Protection Domains

From the application's perspective, the call to `transfer` that is made in a message call is implicit. The unit for memory protection is the virtual memory segment, which may be as small as one memory page. At times, an application object may wish for a finer grained protection mechanism during communication.

Software fault isolation (SFI) [47, 74, 90] is one means of introducing finer grained protection than per-page. Since the overhead of SFI is in the range of 3 to 200% of normal code executions [47, 66] (depending on whether SFI was compiled in or imposed at runtime), an alternative solution to SFI which is consistent with the Chameleon model is given. When absolutely necessary (for example, during testing), in-host communication is performed in the same manner as inter-host communication and will result in an expensive copy to a different block of memory. This would ensure that unexpected access to passive objects will not be granted, but at the expense of a performance degradation that is very similar to traditional inter-process communication.

Another issue with protection is the level of which that is applied to brokers. AbstractCPU works in a protection domain that is the combined set of all other protection domains. Because AbstractCPU defines the environment for all brokers, every broker has access to any protection domain. The issue of trust resurfaces. A broker, however, is able to apply a protection domain to itself as necessary, for example, for testing.

5.1.5 Summary

Mungi researchers reject the idea of "passing by reference" passive objects among active objects because they apparently circumvent the normal protection system, obscure

the protection model, and reduce the owner's control over their objects "as access could not be reliably revoked". The Chameleon solution provides the same granularity of memory protection (a memory page), and remains consistent in its design. Issues regarding revoking access rights are typically associated with failed objects, where the state is undefined anyway. The expense of creating new protection domains in Mungi is eliminated in favour of modifying existing protection domains.

The set of memory primitives is small but sufficient to perform dynamic memory allocation and freeing, to implement shared memory blocks, and to implement in-host message passing. The SAS design simplifies and streamlines the implementation of the primitives and of the meta-spaces. In fact, where memory management policies can change during the lifetime of an application, a SAS design may at times even be required. For example, when the meta-space decides to combine the protection domains of two heavily communicating objects, moving one object to a different address range is not always acceptable, possibly due to timing constraints.

5.2 CPU Management

A Chameleon system is comprised of a hierarchy of meta-spaces. The purpose of using a hierarchy of meta-spaces rather than a traditional operating system structure is to provide specialized environments for particular applications, which may have specialized/unusual scheduling requirements. For example, the soft real-time aspect of multimedia has forced changes in current desktop operating system schedulers. Many existing solutions revolve around the notion of a hybrid scheduler that supports both limited real-time scheduling (typically priority-based) and more traditional performance-oriented scheduling. These scheduling solutions are typically difficult to prove correct [1]. Also, the requirements of some jobs may change over their lifetimes. A batch job may exhibit periods of high disk activity, then intensive CPU usage. The meta-space is a

suitable level of granularity to provide different scheduling policies. Therefore, an alternative to a hybrid scheduler is possible, and even necessary. The purpose of this section is to discuss distributing scheduling operations to the hierarchy of meta-spaces.

Hierarchical, or multi-level scheduling is applied to support dissimilar scheduling algorithms and object requirements, and possesses the following advantages:

- Support for multiple distinct scheduling policies on the same or different processors to deal with different and unforeseen problem domains [75].
- *A priori* scheduling information may not be available for the general scheduler, but can be available at creation time of a user-level scheduler [48].
- Where it is impossible to program into a system all desirable scheduling policies and to have them work together at the same scheduling layer in some meaningful fashion [48], hierarchical scheduling can be used to break the issue into smaller, manageable problem domains.
- Hierarchical scheduling provides hierarchical control over processor resource usage through different logical or administrative domains in a system, such as users, groups, individual processors, and threads within a process [75].
- New scheduling algorithms can be tested on an existing system with full debugging and performance measurement support, *but without impact on other, essential parts of the system or even requiring a stand-alone system for the experiment.*
- Dividing scheduling solutions among fine-grained "scheduler" objects will inevitably result in simpler collections of code which will be easier to maintain, debug, and prove correct, than a hybrid scheduler.

Note that a hierarchy of cooperating schedulers in no way guarantees a particular object's schedulability, as each object can be comprised of many jobs that have to be scheduled with the other objects in the system.

There has been previous work in hierarchical scheduling. It was initially investigated in Apertos, but a hierarchy of policy objects providing scheduling information to one global scheduler was adopted (but not implemented) instead [24]. Further, there were no timer interrupts in Apertos and an active object's execution was non-preemptable, making real-time guarantees impossible to fulfill [41]. A hierarchical scheduling system in the PM operating system was discussed in detail in [76]. Work experimenting with schedulers in user space has been done using BirliX [75]. 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 the investigation of interface requirements of schedulers. Fluke defines a framework where arbitrary threads can act as schedulers for other threads [1] to maximize CPU utilization. It requires schedulers to work together by donating and sharing time-slices.

5.2.1 Scheduling

In some respects, scheduling meta-spaces and active objects within meta-spaces is not different from scheduling applications with collections of threads. That is, the meta-space must be scheduled before an active object that it supports can execute, just as an application must be scheduled before one of its threads can execute. This design has had various names, such as task trees, thread trees, and even thread groups as found in Java [22]. However, in present designs [77], the same scheduling principles are often applied by only one scheduler that manages all processes and threads. Even in systems where a scheduler can be defined within an application domain, there is only one main system scheduler, so scheduling applications can remain relatively easy. In Chameleon, a more generic solution must be defined.

Meta-spaces have to be scheduled amongst each other, just as applications have to. The `resolveBroker` method in the broker interface serves this purpose. This

method is invoked when the currently executing broker has no active object to execute, or during preemptive scheduling. This operation was discussed in Section 4.3.3.

Meta-spaces are allowed to use different scheduling algorithms in order to provide the services required for the objects and meta-spaces they support, but must specify its requirements to its supporting layer. Each meta-space manages itself but is scheduled as an entity by its meta-space.

This flexibility introduces a significant problem for the root scheduler, which must somehow manage possibly very dissimilar scheduling algorithms. A policy-free scheduling technique, such as that employed by RT-Mach [9, 78], is chosen to address this issue. The base, or primary scheduler simply provides time slices that can be pre-allocated to dependent meta-spaces and objects whose policy requires this. When an object (or a scheduler) requests to be scheduled, it specifies the periodicity (if any), worst case execution time, etc. The primary scheduler then attempts to add it to the schedule in a feasible manner. If successful, then the requesting object or scheduler has complete control over its time slice allocation, a necessary assumption for some scheduling algorithms, such as Borrowed-Virtual-Time (BVT) [77]. If the object cannot be added to the existing schedule, then this failure is communicated to the requesting object so that alternative actions can be taken. Remaining time slices are applied to other, generic meta-spaces without real-time requirements in a feasible manner.

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 dependent meta-spaces to guarantee the implementation of algorithms, such as earliest deadline first, rate monotonic, or least laxity real-time scheduling, etc.

5.2.2 Object Sharing

Apertos [5, 15, 39] and PM [10, 44] both report advantages of sharing the management of meta-objects (or service objects) among different meta-spaces or meta-layers. However, there appears to be no operating system based on a meta-hierarchy architecture that actually *has implemented* this feature. In Chameleon, service objects are *not* allowed to exist in more than one meta-space simultaneously, and we surmise that it wasn't implemented in other systems because the difficulties that arise far outweigh any benefits.

In Chameleon, it is impossible for an object to exist in more than one meta-space. Each active object has exactly one locus of execution. Each locus of execution is associated with and maintained by exactly one broker, where the broker defines a meta-space. If the design of an active object is modified so that more than one locus of execution may be associated with it, then brokers sharing the object have to be aware of each other and coordinate scheduling of the object to avoid a host of issues, including synchronization and state. If the design of a locus of execution is modified such that more than one broker can be associated with it, then other issues, such as which broker will handle the receipt of a message, arise. In Chameleon, inter-meta-space communication has overheads associated with it that are low enough to negate any advantages that may be found in allowing meta-spaces to share the management of active objects, in light of these complications.

5.2.3 Interrupt Handlers

A hardware abstraction object will exist to abstract each hardware service or device, such as the network card, or hard drive. These objects must deal with hardware interrupts in an asynchronous manner. When an interrupt occurs, a method of a particular object is invoked. Only one method is typically allowed to be registered with a particular

hardware event, contrary to the mechanism and philosophy in the Spin system. The exception to this case is when multiple devices can share the same interrupt, which can be the case on the Intel PC platform.

Interrupt handler methods must be fast enough so that they do not adversely affect overall performance, but are long enough to do something useful. For this reason, an interrupt handler executes in CPU kernel/supervisor mode and is invoked as a simple function call from AbstractCPU (which is not re-entrant and executes with interrupts disabled). Therefore, any interrupt handler method must act within the same confines and rules as any broker, and I/O space and interrupt handlers are not protected from one another. Typically, the interrupt handler is expected to only perform simple verification and then (if necessary) send a message notifying the schedulable portion of the active object. Any message introduced by the interrupt handler method must necessarily continue via AbstractCPU's broker coordinator algorithm to the broker supporting the interrupt handler active object. Then, if necessary, the preemptable active object may choose to implement an event-driven system similar to Spin's spindles.

The interrupt handler active object is supported by its broker. However one method in the active object is executed directly by AbstractCPU and thus executes in the same manner as any broker. Therefore, execution rules applied to the interrupt handler method may be different than those applied to the remaining active object.

5.2.4 Summary

Hierarchical scheduling corresponds nicely with the meta-hierarchy employed by Chameleon, and provides many advantages over traditional scheduling techniques. The design also supports schedulers that may execute in user mode. However, low-level primitives again have to be carefully designed to avoid limiting the adaptability of the system as a whole, and introducing constraints that may constrain the design of a specific

meta-space. As such, policy-free scheduling technique is used in the lowest-level scheduler.

Many different forms of scheduling have been identified in Chameleon. In this process, it has been shown that hierarchical scheduling requiring context switches recursing through the meta-hierarchy have been eliminated in favour of direct method calls via the broker interface. This is a vast improvement over other conceptual models using the meta-hierarchy.

Chapter 6

Communication

A Chameleon system is composed of active objects that use brokers for communication. Communication in Chameleon can be classified into three basic categories:

- communication between an active object and its meta-space;
- communication with *local* active objects – other active objects supported by the same meta-space; and
- communication with *remote* active objects – active objects supported by some other meta-space which may or may not be located on the same machine.

An essential goal in Chameleon is to keep all forms of communication consistent from the perspectives of the requesting and target objects. Brokers and service objects exist to identify and correctly handle each form of message send. However, any consistency must not impose undue restrictions on meta-space policies or implementations.

Another goal is to clearly define how inter-meta-space communication takes place. Although this form of communication is critical for operating system designers using a meta-hierarchy, it has not been properly addressed in other research, probably due to the difficulty in keeping the communication semantics consistent from a variety of perspectives.

This chapter ends by unifying the concepts of communication and migration, and shows how all objects in the system can benefit from unified interfaces.

6.1 Public Interfaces and Message Classes

An object needs a public interface in order for another object to communicate with it[5]. Public interfaces may be declared in an IDL (interface definition language). An IDL is used to remove any programming language specific assumptions that would prevent the object from being used by other objects coded in other languages. An IDL is not necessary for small passive objects that are not intended to interface with objects written in different programming languages.

Each broker defines a dictionary of messages that it accepts, which can be considered a public interface into (or API for) its meta-space. There are also message categories for interrupts and exceptions and for inter-broker communication. Every possible event that can go through AbstractCPU belongs to the message hierarchy. Brokers may use both the message object type information and member variables to determine appropriate actions upon receipt of a message. The actual class hierarchy is given in Appendix A.

6.2 Proxy Objects

Using message objects directly can mean tedious programming for objects performing a lot of communication. Message set-up and clean up is generally simple, repetitious, and error-prone code, so *proxy objects* are used to permit the representation of all remote object references as local points, and thus hides the associated complexity from the caller [79].

---

[5] Here, a public interface only refers to methods which are accessible by some other object, and should not be confused with meanings of Java keywords published, public, etc.

A proxy object handles details such as locating the active object that it represents. This means that if its active object cannot be located, then it may fail. Alternatively, the proxy object may choose to instantiate the active object that it represents. The search for its active object may traverse meta-space or even machine boundaries. The client active object using the proxy object is expected to determine the bounds of the search, and/or determine where a new target active object should, or can, be created.

In Chameleon, a proxy object is typically a passive object that exists in the requesting active object's protection domain. Proxy objects handle messages, and each message actually has two return values:

- There is a return value for the request that the message represents. That is., if an object asked for the handle to another object for communication (name look-up), then the handle would be the return value. The proxy object is unconcerned with this value.
- The other return value is considered a system return value and is a code representing the success or failure of system's ability to process, or handle the request. This value is always set upon return of the call and may indicate whether the other return value is actually valid or not. The proxy object is interested in this value and may deal with it appropriately or may return it to the client object.

This approach is used in Microsoft's COM system [21] and has proven useful in Chameleon, too. It separates the success/failure of the call from the success/failure of making the call in a manner that is practical and easy for the client object to handle. Proxy objects may return the system return value (if failed) via a number of techniques, such as structured exception handling, generally dictated by the programming language of choice.

6.3 Object/Meta-space Communication

Communication between an active object and its meta-space provides a basis for all other types of communication, so it is discussed first. The sender field in a message object is needed by AbstractCPU to identify the broker. Marshalling methods associated with the message object are also needed to transfer access to the message across protection domains. Marshalling methods are actually required for all objects in the system, and are defined in detail in the following sections. The requesting object does not use the receiver field for the message. This is because the receiver of the message was the object's meta-space, not a particular object. The rule here is that when a message is destined for an object's meta-space API (abstracted by a broker which does not have a locus of execution), the receiver field must be assigned the value 0, or NULL. The broker is then free to use this field if and as it needs to, to forward the message to an appropriate service object.

6.4 Inter-Object Communication

Not all messages that an object wishes to send are intended for its meta-space. For example, when one active object wishes to invoke a method of another active object, it must somehow communicate that request via its broker. Here, the Receiver method in the message will identify the target object instead of being left as NULL. Since the Receiver field is non-zero, the broker can recognize it as a message not destined for its API. This technique is important for primarily two reasons:

- The concept behind a meta-space is that it is unobtrusive. That is, it performs its actions in such a manner that supported objects need not be aware of its existence.
- Supporting this implicit message send removes more broker-specific dependencies from the application layer because no specifically defined '*Send Message*' action is common to all broker dictionaries.

The receiver's value is important because the target object may share some of the same dictionary as the requesting object's meta-space.

6.5 Communication Beyond The Local Meta-space

When an active object sends a message that is to be delivered to another object, it is seldom concerned as to whether the target object is supported by the same meta-space or a different meta-space. For example, a bitmap editor may want to modify an image that needs to be retrieved from a database of images that is supported by another meta-space. Also, a local namer service object may need to communicate with another namer object to locate an active object residing in another machine. This section defines how this type of communication takes place.

When a broker encounters a message with a receiver that is not managed by it, the message can be forwarded to the broker managing the receiver via the `brokerMessage` method call in the broker interface. This method was originally introduced in the Section 4.4 for inter-broker communication, for example to attach and detach dynamically created meta-spaces. This type of communication is just another form of inter-meta-space communication that is handled in this consistent manner. The target object's broker, however, is only able to queue the message for later handling. It has to wait for its meta-space to be scheduled via the broker interface's `resolveBroker` method.

If communication involves the network, then the message will have to be packaged, and then handled by the network active object. An active object acting as a proxy, representing the target object on the remote machine can handle this operation.

6.6 File Systems, Communication, and Migration

For consistency reasons, many research SAS projects [67, 68, 81] are intent on eliminating the file system and sharing the virtual address space across a distributed

environment. In contrast, Chameleon is unique in that the file system is retained. Distributed shared memory (DSM) [80] is not supported in Chameleon primarily to ensure more autonomy among machines and to increase a system's fault tolerance. Consistency is achieved from the requesting object's perspective in the manner in which objects are moved to/from the file system, how objects are migrated to a remote machine, and how inter-object communication is performed.

No alternative techniques are needed in Chameleon for communications with a traditional system, as would be in a system based on DSM. For example, [68, 72] do not outline how Mungi processes would interact with a traditional UNIX system. Because DSM is not used in Chameleon, this model doesn't have to be augmented other than with the specification of a common communication protocol.

6.6.1 Object Marshalling and Serialization

In traditional systems where structures are elementary (contain primitives like characters, integers, etc., but not pointers to other structures or other parts of themselves), moving structures from one protection domain (or machine) to another is a straightforward copy. Furthermore, when structures are composite (that is, do contain references to other structures), these pointers are relatively easily identifiable and serialized across protection domains. However, in the object-oriented world where the object is supposed to be a protected entity and where its make-up and size are hidden, references to embedded objects will not be accessible by an outside entity unless the object-oriented paradigm and programming language rules are circumvented. This problem of embedded pointers is present in any system using some form of IPC or object serialization; the object-oriented model only complicates the issue by imposing additional rules. In fact, this is a primary reason for using shared distributed memory in some systems [81].

Current commercial application approaches refer to the task of saving an object to permanent media as *object serialization* [82]. During serialization, a particular method or methods in the object's interface are invoked to place an object onto a serial data stream so that the object can be recreated at a later time. Version information and each member variable are individually sent to a serial data stream. The object's implementation is allowed to evolve over time, and language rules do not have to be circumvented. However, the cost of the operation is directly related to the size and complexity of the object.

In Chameleon, object serialization primitives are modified so that rather than placing each member variable into a data stream, the start address and size of the member variable is placed into the stream. This stream is analyzed to either modify a target object's protection domain or identify the object data that needs to be saved to permanent media. The size and complexity of this stream is then directly related only to the complexity, and not the size, of the object being serialized. This point is important because large, elementary objects such as those encapsulating multimedia data streams can be handled cheaply, where explicit copying that may otherwise be performed, is prohibitively expensive. In fact, the serialization mechanism normally used by applications has been replaced with a marshalling mechanism that will serve to drive serialization of the object when necessary. Issues such as member variable, or even code, relocation are resolved when the object is restored into memory. For example, arrays of pointers may be translated to arrays of indices. Version information remains the responsibility of the object being serialized.

When communication is performed across a network, or when a document object is saved to permanent media, some form of serialization is required anyway, even when distributed shared memory is used. In addition, these routines can be applied to active objects, or to complete meta-spaces, for migration. The serialization mechanism using

the marshalling routines is unconcerned over the type of object being marshalled and/or serialized, or why the operation is taking place. It is only concerned whether:

- the object is active or passive (to call a function or create a message);
- the object is in an appropriate state (not preempted where the state may be in flux); and
- that the object that these operations are being applied to, supports the appropriate interface.

This migration and serialization mechanism is equally applicable to traditional, multiple address space operating systems. However, the option of using the marshalling mechanism to modify protection domains without remapping its contents would be lost.

6.7 Costs Associated With Communication

In Chameleon, all events are abstracted as messages residing in memory, and these messages are sent to protected entities: objects (both passive and active). There are two costs associated with communication in Chameleon:

- A traditional design can make a very efficient use of registers when entering supervisor mode for simple API calls [7, 83]. In Chameleon, communication is only performed with message objects that are located in memory (the heap or the stack), so messages must be repeatedly constructed and destroyed - or at minimum re-initialized. These operations may require memory management code to be invoked to allocate memory, which in turn may cause more messages to be constructed and used.
- The use of active objects to represent hardware resources requires using message objects that may be queued. For example, in a traditional OS design, the kernel may treat console memory as shared, so writing to the screen is straightforward. In Chameleon, the console memory is managed by an active object with a specific interface, so a message must be constructed and sent. Then, the console object

must be scheduled to receive and interpret the message and then proceed in a similar manner to return any error codes, etc.

These two issues became evident during the development of Chameleon. Even though inter-object communication performance and scheduling are much more efficient than typical traditional designs, the cost of simply writing a text string to the screen in Chameleon is higher because, regardless of the sizes or contents of the messages, there is simply more communication and scheduling taking place. These two issues are often overlooked or improperly identified in other research, where "poor performance" is used as motivation to move server code from application processes to kernel threads without a proper examination of what performance is being measured [12, 56]. As a result, even if a particularly efficient design for inter-object communication is employed at the system level, communication may still be cited as the performance bottleneck.

This is not a problem unique to Chameleon, but is exhibited by any system favouring protected entities over shared structures. It is argued that the gains in extensibility, adaptability, and flexibility, as well as the ability to define systems that are optimal for particular applications, are worth these expenses during communication. This is especially true as when a traditional design grows more complex and more mechanisms have to be introduced to provide various levels of protection and synchronization to the shared structures to ensure correctness of operation.

6.8 Summary

Methodologies are needed in any system to transfer message data from one protection domain (or machine) to another. Where traditional and even research systems provide a variety or primitives and techniques for these tasks, Chameleon has unified this procedure. From an application object's perspective, the means for transferring data to another protection domain, to a hard drive, or to another machine, have been unified.

These methodologies work equally well for migration of object code and data, and for messages used for communication.

Chapter 7

Performance Results

In the previous chapters, the Chameleon model, its components, and implementation issues were presented. Differences in design, along with advantages and disadvantages, between Chameleon and other designs were also pointed out. In this chapter, the implementation of the operating system is reviewed and tested. In particular, we show that the performance draw-backs of meta-space structured operating systems – as illustrated by the performance of Apertos – have been successfully overcome by the Chameleon design and architecture.

7.1 Execution and Compilation Environments

The Intel x86 architecture was chosen for the test-bed operating system. The hardware is cheap and plentiful, many drivers are available for a host of hardware combinations and options, and many small, experimental operating systems are available on which to base the development of the Chameleon system. The operating system is built using the latest GNU compilation tools for Microsoft Windows and MS-DOS. The programming language chosen for development is C++. It has the necessary object-oriented paradigm abstractions, the compilers are efficient and relatively bug-free, and the ability exists to override language conventions and security when warranted. Assembler code is only used where necessary to minimize platform specific source code. The executable code was built for a Pentium-based processor with all performance optimizations turned on during compilation.

An experimental operating system called COSMOS [84] was chosen as a basis for development. COSMOS is a tiny, elementary system using a traditional kernel design that provided the basics needed for multi-processing, user interaction, and test/example applications. Building Chameleon from a more traditional structured operating system design enabled the system to be built much faster, since start-up code, context switching code, and device drivers were already available in working form[6]. In addition, it allowed for the observation of many differences between Chameleon's object-oriented design and a more traditional design. These conceptual differences, such as re-entrant kernels vs. a single thread of control in AbstractCPU, have already been discussed in previous chapters.

Revisions to the COSMOS source codes have been extensive, and very little of the original system exists in the Chameleon source. Portions of the device drivers, the boot code, and some of the assembler code remain unmodified. All of the code implementing the system's unique characteristics, being the entire C source, has been replaced with new C++ source code for Chameleon. Application source code has received virtually no modifications, and changes were limited to the support libraries. Both systems boot from MS-DOS so that no hard disk drivers are needed. Once the operating system begins executing, MS-DOS is completely discarded so that the system executes on bare hardware.

Chameleon has three different broker implementations: PrimaryBroker, ApplicationBroker, and RTBroker. These brokers were discussed in Section 4.4.1. The rest of the Chameleon operating system can be summarized as AbstractCPU, some device

---

[6] Developers for the Spin operating system reported that a lot of time was spent discovering, then debugging errors in supplied code. This project was no different, and some of the hardest-to-find bugs were buried in code we assumed was verified and correct.

drivers and service objects, and objects used only for timing purposes. Each broker has its own locus of execution class and its own scheduler implementation. Two brokers (ApplicationBroker, RTBroker) share the same basic API, where PrimaryBroker implements another. Details are outlined in Appendix C.

The operating system (both Chameleon and COSMOS) and its applications are first built as separate executables, and are then placed in the same file. The operating system treats the loaded file as a RAM disk containing separate application images. This rule guarantees that the compiler or linker cannot make any optimizations to the system, as may be the case if the application and operating system were linked together. Avoiding this potential for optimizations ensures that the model is representative of true costs for better comparisons with other research.

7.2 System Start-up and Run-time Environment

Even in its current state of development, Chameleon is capable of adapting to fit application needs. The system's state at start-up is substantially different from that at any subsequent time. The start-up code begins by instantiating a PrimaryBroker object (with AbstractCPU) and its service objects. Then, an ApplicationBroker object is created and registered with PrimaryBroker, and any applications found are extracted and set up to execute. Once all application objects are instantiated and registered with ApplicationBroker, the system is allowed to start.

When PrimaryBroker first schedules the service objects, they initialize themselves and enter a wait state to receive notifications of interrupts or requests from application objects. Applications start up differently. They immediately print messages to their respective consoles, and the real-time clock active object immediately requests to be migrated to a different (real-time) broker (RTBroker). When RTBroker is instantiated, it

registers itself with PrimaryBroker too. During migration, all management of the clock object is passed to the new broker, and the clock's locus of execution is replaced with another. These differences in active object start-ups and meta-spaces clearly illustrate that the Chameleon architecture supports active objects with substantially different expectations from their run-time systems, and that multimedia applications with soft real-time constraints can be served along-side traditional applications.

Other active objects/applications may request to be migrated to another instantiation of ApplicationBroker upon user intervention. When these requests occur, new brokers are instantiated as necessary. Also, if an object migrates from a broker and the broker no longer supports any active objects, the broker will automatically detach itself from the system and perform any necessary clean-up, including having itself deleted. Brokers/meta-spaces are repeatedly created and destroyed upon application requests at run-time.

7.3 Performance Measurement: Configuration, Tools, and Techniques

A variety of tests have been performed to verify correctness of operation of the operating system, measure costs of various operations, and to compare Chameleon against other operating systems. All tests were conducted on two configurations of Chameleon; the first configuration used one application-specific broker (ApplicationBroker) to manage all applications. The second configuration distributed the same applications among 5 instances of ApplicationBroker, all registered to PrimaryBroker. For both configurations, PrimaryBroker managed 5 active objects, being the 'idle' object, interrupt handlers and device drivers, and test objects. The real-time clock was disabled for timing purposes. Testing occurred without any reboots; the system was reconfigured at run-time.

Table 7.1 Comparison of Simulator and Hardware Timings

Operation	instructions (simulator)	cycles (simulator)	cycles (hardware)
1 NOP instruction	1	1	1
5 NOP instructions	5	5	1
count to 1 million	2000002	2000002	2000002
getpid (PrimaryBroker)	412	691	611
getpid (ApplicationBroker)	344	590	622
intra-meta-space IPC, 1 ApplicationBroker	2269	3469	2785
intra-meta-space IPC, 5 ApplicationBrokers	2285	3485	2785
inter-meta-space IPC, 1 ApplicationBroker	2398	3731	4599
inter-meta-space IPC, 5 ApplicationBrokers	2480	3810	4614

All hardware tests and measurements were performed on a 655Mhz Pentium III computer with 768MB RAM. The hardware platform provided real-world numbers where costs and gains associated with memory, the cache, pipelining, etc. could be incorporated. Pentium series and newer CPU's have an onboard 64-bit cycle counter that constantly runs parallel to all execution and does not impact performance. This cycle counter was used to provide the hardware measurements that are presented.

An Intel-based hardware emulation package called Bochs [85] was also used. The Bochs simulator was used for initial debugging and to provide actual processor instruction sequences for analysis. It is a public domain emulation package that is capable of executing most popular Intel operating systems, including various versions of MS-Windows, Linux, and OS/2.

The instruction sequences supplied by Bochs were profiled by another program written for this research because Bochs only provides instruction counts, not cycle counts. Our profiling program applies the symbol files generated by the linker to instruction sequences to trace the flow of execution through applications and service objects executing in user mode, and the AbstractCPU/broker code executing in supervisor mode. In addition, this program generates expected cycle counts for all instructions and

function calls. This information has been analyzed to verify correctness of operation, profile each and every step of the executions, and in some cases to improve the overall performance by identifying inefficient programming[7]. The heart of the profiling program is based on source code from a disassembler to correctly follow instruction sequences. Each individual instruction cost is based on data published by Intel [86].

7.3.1 Hardware vs. Simulator Timings

While simulated and actual timings do not correspond precisely, the trends/ratio between the two are very consistent. Table 7.1 presents some comparisons of both basic and complex operations. Generally, simulated cycle counts are within 25% of actual timings. Discrepancies between measured hardware timings and simulated processor instructions and cycles can be identified as follows:

- Some processor instructions vary in time (cycles) to complete the task, depending on parameters or CPU mode. The simulator cycle count is always pessimistic in this situation, using worst case.
- The simulator does not model the cache and assumes that there is no cost in accessing memory. Given bus bandwidths and memory speeds relative to the CPU clock speed, this is unreasonable. The cache can often compensate for this overhead, but during context switches it may be flushed. This work has also found that object-oriented code typically compiles to very memory-intensive executables.
- The simulator does not account for other hardware costs and features, such as flushing the page table, instruction pipelining, etc.

---

[7] The original RTTI implementation had to be substantially revamped to obtain performance numbers that were not adversely skewed.

Given that the same CPU can exhibit very different MIPS measurements due to bus width and speed, cache sizes, memory speed, etc., these differences are considered very acceptable. So, actual hardware cycles are presented for comparison data against other operating systems. Hardware timings in microseconds are given for completeness, but do not account for different CPU speeds. Cost breakdowns for individual operations are based on simulated cycle counts because they serve as a consistent standard to work from that do not include hardware variations such as flushing the instruction pipeline during timings.

7.3.2 Comparative Data

The following tables in this chapter present much data from various operating systems executing on different hardware for a direct measure of Chameleon's performance. The comparative data is provided by [56, 7, 52, 68, 57, 12]. There are many costs that vary across processor families. For example, an i486 trap is very expensive, but SPARC processors have to deal with register windows. Costs of memory management add another level of difficulty for comparison. Cycle times are presented in addition to actual timings to remove processor speeds from the equation.

Differences in timings for the Mach and L4 micro-kernels (given later in tables) illustrate the important observation that software designs (in this case micro-kernels) will not necessarily exhibit the same changes in performance after porting. IPC for Mach on an Alpha processor uses slightly more cycles than Mach on a Pentium processor, whereas L4 is considerable faster on the Alpha processor than a Pentium. Also, specific details are often missing regarding how the comparative data is gathered and what the comparative data is measuring. For example, information as to whether (necessary) stub code is included in the IPC timings, is often not available. Restricted vs. full IPC semantics are not defined. Given all of these variables relating to testing environments, hardware, and methodologies, the comparative data is only sorted into in alphabetical

Table 7.2 Comparison of **getpid** *(or equivalent)*

System	CPU, MHz	Cycles	Time
Aegis/Exokernel	R3000, 16.7	58	3.5 μs
Apertos	R3000, 20	1420	71 μs
DEC OSF/1	Alpha 21064, 133	665	5 μs
Irix	Alpha 21064, 133	3192	24 μs
Linux	Pentium, 133	223	1.68 μs
Linux executing on L4 u-kernel	Pentium, 133	526	3.95 μs
Linux executing on Mach (in-kernel)	Pentium, 133	2050	15.41 μs
Linux executing on Mach (user)	Pentium, 133	14710	110.60 μs
Mach	Alpha 21064, 133	931	7 μs
Mungi	Alpha 21064, 133	612	4.6 μs
Spin	Alpha 21064, 133	532	4 μs
Ultrix	R3000, 16.7	562	33.7 μs
Chameleon (PrimaryBroker)	*Pentium III, 655*	*611*	*0.93 μs*
Chameleon (ApplicationBroker)	*Pentium III, 655*	*622*	*0.95 μs*

order, and is presented only to furnish general trends in designs and formulate basic observations.

For an operating system intended to support multimedia data streams, the primary concerns are interrupt handling overhead, inter-object communication performance given small and large amounts of data, and the cost of scheduling objects. The tests carried out reflect those concerns. A summary of these results is given in the following sections.

7.4 getpid Measure

A basic means of comparing systems is the overhead to simply enter supervisor mode and return to the executing thread of control, i.e. the basic cost of an interrupt handler. The UNIX `getpid` routine is considered the shortest standard repeatable operation to test this flow of control. This operation doesn't require a reschedule operation. Table 7.2 presents a summary of comparisons of this operation on a variety of architectures. Data for Chameleon are italicized.

The variation of performance even within processor families on this simple call is surprising. This operation is considered expensive in Chameleon, since AbstractCPU's entire algorithm still has to be executed. Most other systems will not have any

Table 7.3 Comparison of 1-byte IPC Performance

System	CPU, MHz	IPC cycles (round trip)	IPC time (round trip)
Aegis/Exo-kernel	R3000, 16.7	74	4.4 µs
Amoeba	68020, 15	12000	800 µs
Apertos (intra-meta-space)	R3000, 20	17440	872 µs
DEC OSF/1	Alpha 21064, 133	112385	845 µs
DP-Mach	80486, 66	528	16 µs
Exo-tlrpc	R2000, 16.7	53	6 µs
Irix	Alpha 21064, 133	59850	450 µs
Kea	80486, 50	4350	87 µs
L3	80486, 50	500	10 µs
L4	Alpha, 433	90	0.2 µs
L4	Pentium, 166	242	1.5 µs
L4	R4600, 100	200	1.7 µs
Lipto	68020, 16.7	3600	240 µs
LRPC	CVAX, 12.5	981	157 µs
Mach	80486, 50	11500	230 µs
Mach	Alpha 21064, 133	13832	104 µs
Mach	R2000, 16.7	3168	190 µs
Mungi	Alpha 21064, 133	1729	13 µs
Opal	Alpha 21064, 133	15029	133 µs
QNX	80486, 33	2508	76 µs
Spin	Alpha 21064, 133	13566	102 µs
Spring	SparcV8, 40	220	11 µs
SRC RPC	CVAX, 12.5	5800	464 µs
Chameleon (intra-meta-space, 1 ApplicationBroker)	*Pentium III, 655*	*2785*	*4.25 µs*
Chameleon (intra-meta-space, 5 ApplicationBrokers)	*Pentium III, 655*	*2785*	*4.25 µs*
Chameleon (inter-meta-space, 1 ApplicationBroker)	*Pentium III, 655*	*4599*	*7.02 µs*
Chameleon (inter-meta-space, 5 ApplicationBrokers)	*Pentium III, 655*	*4614*	*7.04 µs*

comparable overhead. Despite this known overhead, Chameleon still presents "middle-of-the-road" performance.

7.5 IPC Measure

IPC, or inter-process communication is a standard performance measure. Table 7.3 presents the cost of IPC for some well-known research and commodity systems. Data for Chameleon are italicized. 1-byte IPC is presented because most all of the comparative data used this as a standard test.

The table shows that while Apertos is the least efficient design presented for research systems, Chameleon is "middle of the road" between small high performance micro-kernels that require a lot of support libraries (such as L3 and L4) and large, mature commercial kernels such as Mach. Note that L4 does not require a scheduling operation during IPC. It is not unique in this respect. The current Chameleon broker policies are to always re-schedule. Considering all the variables in processor type, hardware configuration, etc., these findings are very promising in light of the additional flexibility and extensibility available in the Chameleon model. The communication times that have been presented for Chameleon intrinsically include costs associated with support for migration, a feature not in other systems in the table.

Inter-meta-space communication in Apertos was not implemented so could not be benchmarked. Inter-meta-space communication in Chameleon carries a reasonable overhead over intra-meta-space communication. The simulator data for IPC, presented in Table 7.1, shows a roughly 8% theoretical increase in time incurred when objects that are communicating do not share the same meta-space. Actual inter-meta-space communication in Chameleon, presented in Table 7.3, shows a lot more increase in cost because a change in protection domains has occurred. These figures therefore represent a worst-case scenario.

7.5.1 Effects of the CPU Cache

Hardware timings in Chameleon were performed with the CPU cache enabled, then disabled, and are as follows:

- the cost of a `getpid` operation increased from 611 to 61227 cycles,
- intra-meta-space communication cost increased from 2785 to 395548 cycles,
- inter-meta-space communication cost increased from 4599 to 433212 cycles.

These timings show roughly a 100 times degradation in overall performance. These findings verify simulation vs. hardware data in Table 7.1 where inter-meta-space communication requires a change in protection domains and thus a cache and page table flush.

These timings clearly illustrate an important concern in operating system design today: the disproportion between CPU and memory speeds and utilization of the cache can have more measurable effect on over-all performance than hand-coding assembler code to attain the best possible performance for a given test. Thus, it is reasonable to use a clean, concise design methodology that maximizes extensibility and flexibility and that doesn't exhibit exponential performance decays as the system grows. Table 7.3 shows how Chameleon has struck a comparable level of performance to micro-kernels, but with a high degree of extensibility.

7.5.2 Costs of Copying Memory

Traditional systems will typically copy memory between protection domains during communication. Exceptions to this case are generally introduced specifically for multimedia applications and are not applied during regular communication. The costs associated with traditional communication techniques were measured in Chameleon (with the cache enabled) to indicate the impact that the SAS memory model has on the system.

Basic inter-meta-space communication cost increased by roughly 1500 cycles when dynamically creating, then destroying a message. This operation will be necessary for a composite message object (one which references another object). This operation may not be necessary for an elementary object, but will depend on the specific memory representation of it.

IPC timings on hardware for larger message objects simulating multimedia messages were within the variance of 1-byte IPC measurements. However, the cost of copying only 1000 bytes (using a simple loop) from one message to another is about 3400 cycles. These numbers show that dynamically creating a message, then copying the message to another domain can easily double the costs of IPC. However, 1000 bytes is a relatively small message for multimedia streams, and is in fact small enough to fit in the cache, which was the case for this test. These costs will increase quickly when the message is not in the cache, and cannot fit in the cache. Where page table entries can be copied to the target protection domain instead of copying the data, fewer bytes are copied and cache hits are avoided.

Table 7.4 Costs of Various Chameleon Operations

Operation	Min. Cycles	Max. Cycles	Avg. Cycles
'new' operator (creation of a passive object)	1106	1712	1507
RTTI initialization	29	29	29
Message construction in AbstractCPU for hardware interrupt	49	49	49
AbstractCPU hardware abstraction portion, message introduced by an active object	224	279	273.2
AbstractCPU broker coordinator portion, for message introduced by an active object	77	100	84.1
AbstractCPU broker coordinator portion, for hardware interrupt	93	93	93
Migration from one ApplicationBroker to another	15804	21521	19002

These results are consistent with [60] where transferring ownership of memory pages instead of copying increased communication performance by a minimum of 240%. Opal [67] saw even higher improvements, in the range of 340%. It is clear that a SAS memory model can provide substantial gains in IPC.

7.6 Costs of Various Operations

In this section, the costs of various different operations are presented to put the costs of the broker interface into perspective. This data was garnered from the simulators. After these operations are outlined, a complete cost break-down of intra-meta-space IPC is given.

The values presented in Table 7.4 are constant regardless of the number of brokers in the system. The fact that the core operations in the broker interface and AbstractCPU algorithms are constant in cost is very important in meeting soft real-time expectations, and an important scalability factor of Chameleon.

The 'new' operator measures the cost of Chameleon's dynamic memory allocator. The variance is based on the length of time it takes to search the free memory pool for an appropriate memory block. This test has been included to put the other costs in perspective.

The construction of a message object in AbstractCPU measures the cost of adjusting the stack, and calling the message's constructor method. The cost of RTTI initialization and message destruction has also been included. This value is constant because the set of messages created by AbstractCPU to represent hardware events is a closed set, and each message happens to cost the same amount.

The cost of the hardware abstraction portion of AbstractCPU is quite deterministic. It includes the time to enter, then exit supervisor mode. The cost of the

Table 7.5 Cost Breakdown of Simulated IPC

Operation	Cycles	Instructions
Construct, initialize, then destroy message by caller	115	56
AbstractCPU hardware abstraction portion	702	325
AbstractCPU broker coordinator portion	231	180
PrimaryBroker	*2229*	*1631*
• PrimaryBroker::saveContext	75	42
• PrimaryBroker::securityCheck	420	357
• PrimaryBroker::resolveTarget (includes scheduling operations)	1515	1067
• PrimaryBroker::resolveMessage	204	153
• PrimaryBroker::restoreContext	15	12
Target object response (construct a return message, etc.)	192	77
Total	**3469**	**2269**

code that can be considered Chameleon-specific is only about one third of the total expense of this operation, though. The remaining costs are specific to the hardware so will be present in any operating system on an Intel platform.

The costs for the broker coordinator portion of AbstractCPU has been divided into two categories:

- where a message was introduced by an active object; and
- where an asynchronous hardware event such as an interrupt occurred, requiring a slightly different path through the algorithm.

These costs are very deterministic, and the overhead associated with the broker interface remains constant regardless of the number of brokers existing in the system.

Object migration to another broker is an expectedly expensive operation. However, one should note that migration in Chameleon has about the same cost as basic communication in Apertos. Apertos didn't actually implement migration so we have no comparison for this task.

Table 7.5 gives the cost break-down of round-trip communication for two objects supported by PrimaryBroker. Note that the actual implementation of this operation is that the server object has to respond to the request, then send another message notifying its broker that it is awaiting for a message. Given this information, we deduce that AbstractCPU overhead accounts for roughly 27% of the overhead in round-trip communication. This percentage includes hardware-specific costs associated with entering CPU supervisor mode that is present in any operating system. The cost of scheduling – determining the next active object to execute - is about one third of total IPC overhead.

7.7 Source Code and Binaries

Table 7.6 gives a comparison of source code and binary sizes. Three snapshots were observed during Chameleon's development: the original structured operating system that Chameleon was developed from (COSMOS), Chameleon with no significant additional functionality, and Chameleon in its current state. A comparison of the first two snapshots is indicative of the basic overhead in using object-oriented technologies and introducing concepts such as classes with well-defined interfaces, RTTI, etc. A comparison between the second and last snapshots indicates the amount of code to introduce more brokers, real-time support, migration, and testing and timing code.

Table 7.6 Comparison of Source Code And Binary Sizes

	COSMOS	**Basic Chameleon**	**Current Chameleon**
Source Code	181KB	302KB	465KB
Operating System Binaries	32KB	56KB	92KB
Application Binaries, Combined Total	122KB	152KB	168KB

Brokers are relatively small in nature. PrimaryBroker is roughly 500 lines long, ApplicationBroker is approximately 1000 lines long. CRTBroker, which inherits from ApplicationBroker, is only 175 lines long. AbstractCPU, which provides a basis for all brokers, is roughly 850 lines of C++ source and an additional 350 lines of assembler code. The actual C++ code for the algorithms using the broker interface is about 250 lines.

The size of the binary images increased considerably over the course of development due to many more statically defined objects used for RTTI, etc.

7.8 Summary

True hardware timings are given for direct comparisons against other operating systems. These measures show that an important goal in this work has been achieved: that Chameleon's performance, using traditional measures, is comparable to current operating system designs. This goal is important to the real-world applicability of Chameleon.

Outputs and data from the simulators have been used to profile and verify correct operations in Chameleon. It has also served to isolate certain costs and benefits associated with the hardware, such as pipelining.

The disparity between CPU and memory speeds has been shown by disabling the cache. Costs associated with flushing the cache during IPC have been isolated by comparing real hardware and simulator timings. These differences show that the effective use of the cache can have more impact on IPC than hand-coding assembler code to attain the best possible performance for a given test. Thus, it is reasonable to conclude that the use of Chameleon's clean and concise design methodology, which carefully chooses designs such as SAS, can maximize extensibility and flexibility and still achieve fine performance measures.

Chapter 8

Conclusions

The primary objective of this work was to support multimedia applications. A high degree of extensibility and a clean object model and design has been attained with very acceptable overhead. There is no redundancy in design, and each component has been carefully chosen or designed to support other parts of the Chameleon in a beneficial manner. The implementation of the Chameleon model has followed and verified all of the aspects in the conceptual design: the object model, AbstractCPU and brokers, hierarchical management of memory and the CPU, and communication and supporting constructs. Chameleon starts as minimal system that is built up in response to user requirements to accommodate the particular applications being executed, including those in the multimedia domain. Most importantly, it can dynamically evolve over time as new brokers and meta-spaces are defined and introduced, and it employs techniques designed to avoid linear or exponential slow-downs in system performance as the system grows.

The object paradigm definitions for particular abstractions, being *active object*, *meta-space* and *meta-object* have been further clarified for this field of research. This work has applied the notion of a *locus of execution* that is object-oriented by design and by application, to active objects. Separating the concept of an object that is *meta* from an object that simply provides a service is also crucial to the basic underpinnings of a *meta* methodology, in that not everything is necessarily *meta*. A clear definition of the object paradigm has also allowed for an interesting concept, called *dynamic class binding* to be used in the AbstractCPU/broker superclass/subclass relationships.

The micro-kernel concept has been reviewed to help define a new concept utilized by AbstractCPU and brokers. The traditional line of abstraction has been moved to one which separates low-level objects abstracting hardware and system-level objects implementing policies. Object-oriented interfaces are applied and leveraged to allow different policies to co-exist at the system level, and provide extensibility and dynamic replacement of virtually every component in the system. AbstractCPU provides an environment for all brokers, which together define specialized environments for all active objects.

Chameleon's broker interface design separates the notion of an object's *dictionary* from its *interface*. This separation is a key aspect of this design, and is a cornerstone to the Chameleon model evolving over time. Designing a suitably abstract broker interface is unique in approach to extensible operating system research, where typically the design focuses on a complete interface. Moreover, the broker design is capable of modelling current constructs such as kernels.

The single address space memory model utilizing multiple protection domains has been identified as ideal for Chameleon. An object-oriented operating system performs far more explicit communication, and the memory model has a direct affect on communication performance. There are also advantages in this design for systems to reduce communication overheads while transferring large amounts of raw data between protection domains, as can be found in multimedia applications. This work has also unified the concepts of communication and migration. Here, from the perspective of the requesting objects, serializing an object to/from permanent storage is performed in the same manner as transferring an object (active or passive) to another protection domain or computer. The application of this memory model is also different than other research designs, in that distributed shared memory is not used.

Hierarchical models for CPU and memory management are clarified. These models remain consistent from any meta-layer. The set of primitives used in memory management is very small, and is applied to both memory allocation and communication. Hierarchical scheduling is described in detail and is used as a means to ensure that meta-spaces can provide policies that are ideal for their applications, but in cooperation with other meta-spaces.

Communication performance, as well as other measures of performance, of the Chameleon model are comparable to micro-kernel designs. Each meta-space can be tailored to the specific needs of the applications it supports. However, a specialty meta-space would remain infeasible if the overhead for inter-meta-space communication is large. The Chameleon model has reduced inter-meta-space communication overheads to such levels that multi-meta-space designs are now feasible. This achievement was accomplished by moving the management of meta-spaces from schedulable active objects to brokers only executing in supervisor mode, and by defining a simple and direct inter-broker communication mechanism. Further, the reduced cost of communication hasn't simply been relocated to (e.g.) stub code, interrupt handling or a binding mechanism.

We conclude that the Chameleon structuring concept is a sound, viable approach to operating system design, suitable for systems that must support multimedia applications alongside traditional applications.

8.1 Future Work

At this time, all Chameleon brokers implement their own schedulers. Schedulers that are replaceable and external to the brokers (implemented as either active or passive objects) would introduce another level of flexibility but may impact performance. Schedulers that are implemented as active objects can also serve to gain a higher degree of concurrency in multi-processor computers.

AbstractCPU uses a simple mechanism to bind to brokers. Brokers also perform inter-broker communication to perform necessary tasks. AbstractCPU and any broker, therefore, are susceptible to failure of other brokers. A recovery mechanism for failed or misbehaving brokers (such a recursion through the broker interface) can be implemented for AbstractCPU and made available to brokers.

A more generic object interface mechanism that is programming language independent is also desirable. This interface mechanism should be more akin to COM or CORBA where objects that support the same basic interface only provides handles to other interfaces. Then, once a handle is available, regular communication can take place. This approach is not necessary for the simulations and will not impede performance, but moves the system a step closer to being ready for commercial use.

The adaptability found in the Chameleon model has been applied to new problem domains such as multimedia. It is now a useful tool for other genres of research and with further development, Chameleon could be used in research in mobile computing for load sharing among stationary hosts with ample computing power and mobile hosts with limited resources. Chameleon has importance here for both fast prototyping and deployment.

References

[1] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, S. Clawson. "Microkernels Meet Recursive Virtual Machines" In *Proceedings of the Third OSDI*, October 1996.

[2] J. Nieh and M. S. Lam. "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications" In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[3] P. Cao, E. Felten, and K. Li. "Implementation and Performance of Application-Controlled File Caching" In *Proceedings of the First OSDI*, pages 165-178, November 1994.

[4] S. J. Mullender, I. M. Leslie, and D. McAuley. "Operating System Support for Distributed Multimedia" In *Proceedings of the 1994 Summer Usenix Conference*, Boston, MA, June 1994.

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

[6] W. Cheung and A. Loong. "Exploring Issues of Operating Systems Structuring: From Microkernels to Extensible Systems" In *Operating Systems Review*, 29(4), 1995.

[7] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. "The Performance of μ-Kernel-Based Systems" In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[8] 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.

[9] 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.

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

[11] 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.

[12] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers and S. Eggers. "Extensibility, Safety and Performance in the SPIN Operating System" In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[13] C. Small and M. Seltzer. "Structuring the Kernel as a Toolkit of Extensible, Reusable Components" In *Proceedings of the 1995 International Workshop on Object Orientation in Operating Systems (IWOOOS)*, 1995.

[14] C. Small and M. I. Seltzer. "A Comparison of OS Extension Technologies" In *USENEX Conference Proceedings*, pages 41-54, January 1996.

[15] 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.

[16] R. W. Bryce. Enhancing Real-time Performance of an Object-Oriented Operating System. Master's Thesis, University of Victoria, 1995.

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

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

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language And Its Implementation*, Addison-Wesley, 1983.

[20] B. Stoustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1991.

[21] R. Grimes, J. Templeman, A. Stockton, K. Watson, G. Reilly. *Beginning ATL 3 COM Programming*, Wrox Press Ltd., 1999.

[22] J. Gosling and F. Yellin. *The Java Application Programming Interface*, Volumes 1 & 2. Addison-Wesley, 1996.

[23] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. "Lightweight Shared Objects in a 64-Bit Operating System" In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1992.

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

[25] R. Lischner. *Delphi in a Nutshell.* Wordware Publishing, Inc., 2000.

[26] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1993.

[27] L. Boszormenyi, C. Weich, and N. Wirth. *Programming in Modula-3: An Introduction in Programming With Style.* Springer Verlag, 1997.

[28] S. Lalis and B. A. Sanders. "Adding Concurrency to the Oberon System" In *Programming Languages and System Architectures*, Zurich, Switzerland, March 1994.

[29] R. Lea, Y. Yokote, and J. Itoh. "Adaptive Operating System Design Using Reflection" In *Proceedings of USENIX Conference on Object Oriented Technologies*, June 1995.

[30] A. Lindstrom, J. Rosenberg, and A. Dearle. "The Grand Unified Theory of Address Spaces" In *5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[31] 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.

[32] S. E. Madnick and J. J. Donovan. *Operating Systems.* McGraw-Hill Computer Science Series. McGraw-Hill Book Company, 1974.

[33] T. Hirotsu, H. Fujii, and M. Tokoro. "A Multiversion Concurrent Object Model for Distributed and Multiuser Environments" In *The 15th International Conference on Distributed Computing Systems*, pages 271-278. IEEE Computer Society, 1995.

[34] L. Wohlrab. "Ruling the Complexities of OS Design and Maintenance Using Object-Oriented and AI Technologies" *Technical Report*, Technische Universitat Chemnitz-Zwickau, 1995.

[35] P. Maes. "Concepts and Experiments in Computational Reflection" In *Proceedings of OOPSLA '87*, 1987.

[36] J. C. Fabre and T. Perennou. "A Metaobject Architecture for Fault-Tolerant Systems: The FRIENDS Approach" In *IEEE Transactions on Computers*, 47(1):78-95, January 1998.

[37] D. B. Orr. "Application of Meta-Protocols to Improve OS Services" In *5th Workshop on Hot Topics in Operating Systems*, May 1995.

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

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

[40] M. Horie, J. C. Pang, E. G. Manning, and G. C. Shoja. "Using Meta-Interfaces to Support Secure Dynamic System Reconfiguration" In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, May 1998.

[41] 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.

[42] B. Gowing and V. Cahill. "Making Meta-Object Protocols Practical for Operating Systems" In *4th International Workshop on Object Orientation in Operating Systems*, pages 52-55, August 1995.

[43] H. Okamura and Y. Yokote. "Customizing Application Object Execution by System Object Downloading in Embedded Operating Systems" *Technical Report*, Sony Computer Science Laboratory Inc., Tokyo, Japan, 1999.

[44] 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 SFB 342, Munich, October 8-9, 1992.

[45] P. Parkyak and B. N. Bershad. "Dynamic Binding for an Extensible System" In *Second Symposium on Operating Systems Design and Implementation*, pages 201-212, October 1996.

[46] R. Grimm and B. N. Bershad. "Security for Extensible Systems" In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 62-66, May 1997.

[47] M. I. Seltzer, Y. Endo, C. Small and K. A. Smith. "Dealing With Disaster: Surviving Misbehaved Kernel Extensions" In *Second Symposium on Operating Systems Design and Implementation*, October 1996.

[48] M. I. Seltzer and C. Small. "Self-Monitoring and Self-Adapting Operating Systems" In *The 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.

[49] P. Druschel, V. S. Pai, and W. Zwaenepoel. "Extensible Kernels are Leading OS Research Astray" In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.

[50] B. Ford, M. Hibler, J. Lepreau, R. McGrath, P. Tullmann. "Interface and Execution Models in the Fluke Kernel" In *3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, February 1999.

[51] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. "Fast Concurrent Dynamic Linking for an Adaptive Operating System" In *Proceedings of the Third Conference on Configurable Distributed Systems (ICCDS '96)*, May 1996.

[52] A. C. Veitch and N. C. Hutchinson. "Kea – A Dynamically Extensible and Configurable Operating System Kernel" In *Proceedings of the Third Conference on Configurable Distributed Systems (ICCDS '96)*, May 1996.

[53] D. R. Cheriton and K. J. Duda. "A Caching Model of Operating System Kernel Functionality" In *6th European SIGOPS Workshop*, 1994.

[54] G. Hamilton and P. Kourgiouris. "The Spring Nucleus: A MicroKernel for Objects" In *1993 Summer USENEX Conference*, June 1993.

[55] A. Chatterjee, A. Khanna, and Y. Hung. "ES-KIT: an Object-Oriented System" In *Concurrency: Practice and Experience*, 3(6): 525-529, December 1991.

[56] J. Liedtke. "On μ-Kernel Construction" In *Proceedings of the 15th ACM Symposium on Operating System Principles*. December 3-6, 1995.

[57] D. Engler, M. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management" *Operating Systems Review*, 29(5), 1995.

[58] B. E. Rector and J. M. Newcomer. *Win32 Programming*. Addison-Wesley Developers Press, 1997.

[59] A. Tamches and B. P. Miller. "Dynamic Kernel Code Optimization" *Workshop on Binary Translation (WBT-2001)*, Barcelona, Spain, September 2001.

[60] A. Sarris and S. K. Tripathi. "Writing an Efficient Device Driver for a Multimedia Teleconferencing System" *Technical Report CS-TR-3614*, University of Maryland Institute for Advanced Computer Studies, Department of Computer Science, March 1996.

[61] H. Moessenboeck. "Extensibility in the Oberon System" In *Nordic Journal of Computing*, 1994.

[62] S. Savage and B. Bershad. "Issues in the Design of an Extensible Operating System" In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, November 1994.

[63] L. van Doorn, P. Homburg, A. S. Tanenbaum. "Paramecium: An Extensible Object-based Kernel" In *5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

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

[65] J. Xu, and D. L. Parnas. "On Satisfying Timing Constraints in Hard-Real-Time Systems" In *IEEE Transactions on Software Engineering*, 19(1), January 1993.

[66] A. Vahdat, D. Ghormley, and T. Anderson. "Efficient, Portable, and Robust Extension of Operating System Functionality" *Technical Report CS-94-842*, UC Berkeley, December 1994.

[67] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. "Opal: A Single Address Space System for 64-bit Architectures" In *IEEE Workshop on Workstation Operating Systems*, April 1992.

[68] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. "Implementation and Performance of the Mungi Single-Address-Space Operating System" *Technical Report UNSW-CSE-TR-9704*, University of New South Wales, June 1997.

[69] T. Wilkinson and K. Murray. "Evaluation of a Distributed Single Address Space Operating System" In *16th International Conference on Distributed Computing Sytsems*, pages 494-501. IEEE, May 1996.

[70] J. Chase, M. Feeley, H. Levy. "Some Issues for Single Address Space Systems" In *4th IEEE Workshop on Workstation Operating Systems*, October 1993.

[71] J. Chase, H. Levy, M. Feeley, and E. Lazowska. "Sharing and Protection in a Single Address Space Operating System" In *ACM Transactions on Computer Systems*, 12(4), November 1994.

[72] J. Vochtelloo, K. Elphinstone, S. Russell, and G. Heiser. "Protection Domain Extensions in Mungi" In *5th IWOOOS*, pages 161-165, Seattle, WA, October 1996.

[73] D. Kotz and P. Crow. "The Expected Lifetime of 'Single-Address-Space' Operating Systems" In *ACM Conference on Measurements and Modelling of Computer Systems (SIGMETRICS 94)*, pages 161-170, 1994.

[74] C. Small and M. I. Seltzer. "MiSFIT: Constructing Safe Extensible Systems" In *IEEE Concurrency*, 6(3), 1998.

[75] 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.

[76] 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.

[77] K. J. Duda and D. R. Cheriton. "Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler" In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99)*, December 1999.

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

[79] P. Druschel, L. L. Peterson, and N. C. Hutchinson. "Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto" In *International Conference on Distributed Computing Systems (ICDCS)*, 1992.

[80] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1994.

[81] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P. E. Osmon. "Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel" In *Processings of the 2nd USENIX Symposium on Microkernels and other Kernel Architectures*, 1993.

[82] G. Shepherd, S. Wingo, D. D. McCrory, and S. Wingo. *MFC Internals: Inside Microsoft© Foundation Class Architecture*. Addison-Wesley, 1996.

[83] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. "Stub-Code Performance is Becoming Important" In *1st Workshop on Industrial Experiences With Systems Software (WIESS)*, San Diego, CA, October, 2000.

[84] http://my.execpc.com/~geezer/os/

[85] http://bochs.sourceforge.net/

[86] Intel Corporation. *Intel Architecture Software Developer's Manual (Volume 2: Instruction Set Reference)*, 1999.

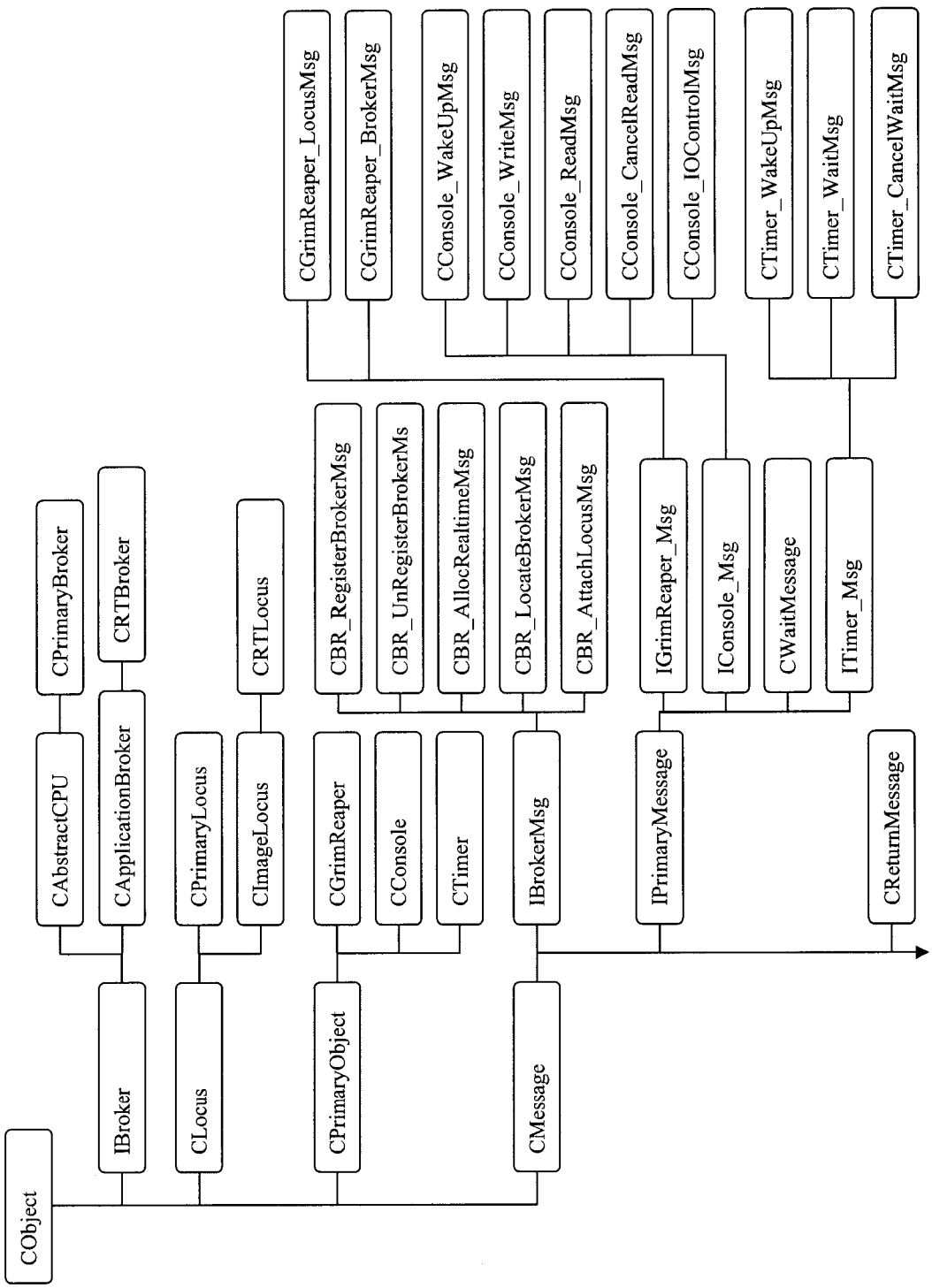
[87] W. O'Farrell and I. Kalas. "Concurrency Support for C++: An Overview" *Technical Report CS-93-03*, York University, North York, Ontario, August 1993.

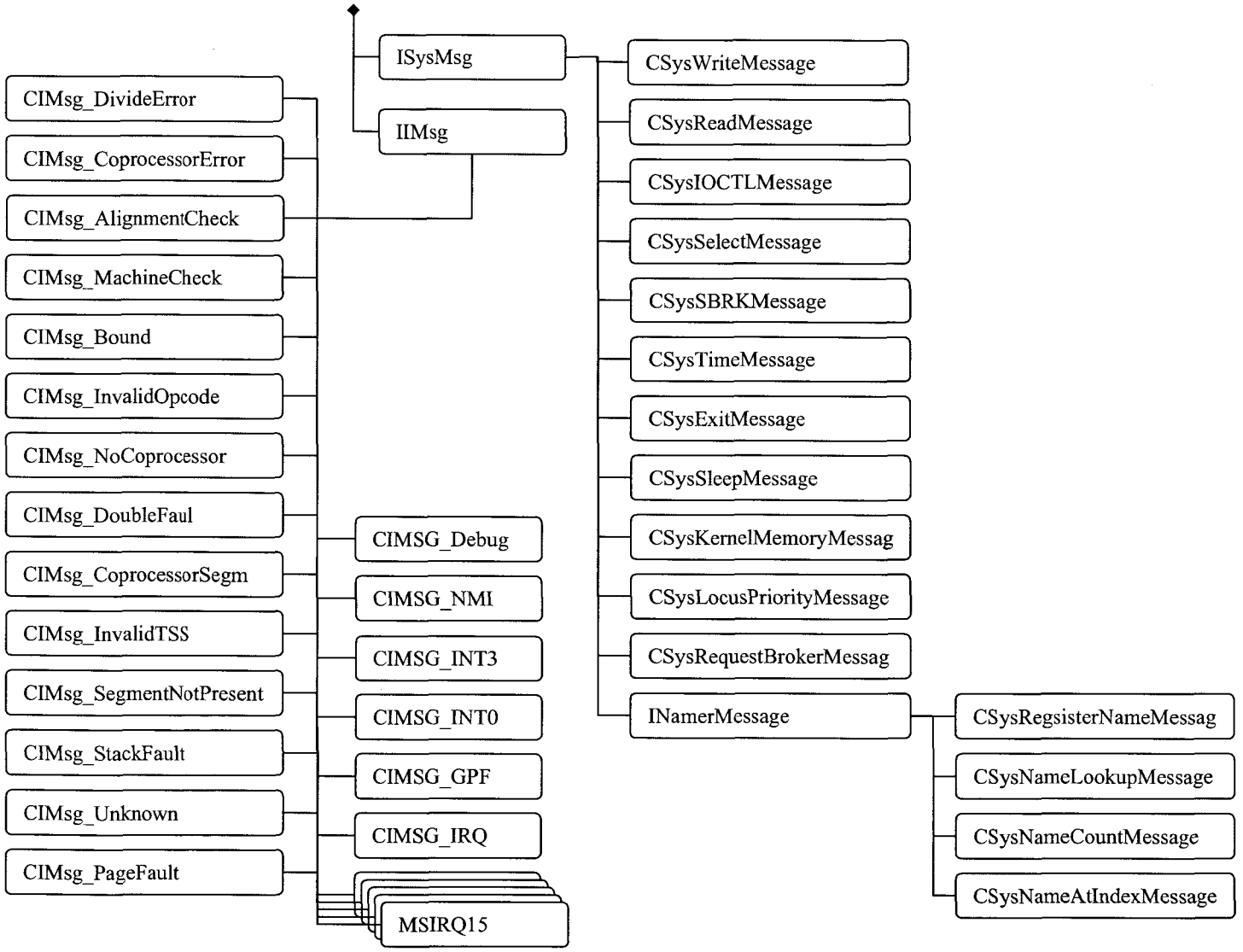
[88] P. A. Laplante. *Real-time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, 1993.

[89] 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.

[90] R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham. "Efficient Software-Based Fault Isolation." In *ACM SIGOPS Operating System Review*, 27(5):203-216, December 1993.

Appendix A: Chameleon Class Hierarchy




ISysMsg
IIMsg
CIMsg_DivideError
CIMsg_CoprocessorError
CIMsg_AlignmentCheck
CIMsg_MachineCheck
CIMsg_Bound
CIMsg_InvalidOpcode
CIMsg_NoCoprocessor
CIMsg_DoubleFaul
CIMsg_CoprocessorSegm
CIMsg_InvalidTSS
CIMsg_SegmentNotPresent
CIMsg_StackFault
CIMsg_Unknown
CIMsg_PageFault
CIMSG_Debug
CIMSG_NMI
CIMSG_INT3
CIMSG_INT0
CIMSG_GPF
CIMSG_IRQ
MSIRQ15
CSysWriteMessage
CSysReadMessage
CSysIOCTLMessage
CSysSelectMessage
CSysSBRKMessage
CSysTimeMessage
CSysExitMessage
CSysSleepMessage
CSysKernelMemoryMessag
CSysLocusPriorityMessage
CSysRequestBrokerMessag
INamerMessage
CSysRegsisterNameMessag
CSysNameLookupMessage
CSysNameCountMessage
CSysNameAtIndexMessage

Appendix B: IBroker Interface Declaration

```
class IBroker : public CObject {
        public:
                virtual HRESULT CoCreate(IBroker *broker) = NULL;
                                                        // used to actually "make" this broker by letting it attach to a managing broker, etc.
                virtual HRESULT CoDestroy() = NULL;     // used to "destroy" this broker by letting it detach from a managing broker, etc., this routine
                                                        // is allowed to fail, and the delete function to the broker shouldn't be called directly
                virtual HRESULT DeleteLocus(CLocus *l) = NULL;
                                                        // given a 'l' needs clean-up

                        // ***** interface that CAbstractCPU uses to talk to brokers in the system
                virtual IBroker *resolveBroker() = NULL;        // called only to and by a broker when somebody isn't sure who can run next (can occur
                                                        // during interrupts, or when the current broker report no active object - nothing to run, a
                                                        // key to letting the broker hierarchy do its thing to defer operations to someone else, but it
                                                        // will start with CAbstractCPU calling that to CPrimaryBroker, which may call another
        protected:                                      // broker, and so-on
                virtual void saveContext(CLocus *l, CMessage *m) = NULL;
                                                        // 'l' is being suspended and this lets its broker do any necessary work to wrap up saving
                                                        // its context.  'm' may be NULL (preemption) or a message that 'l' is introducing
                virtual BOOL securityCheck(CMessage *m, HRESULT *result) = NULL;
                                                        // whether or not a given message can be accepted into the system - m will contain a sender
                                                        // and receiver at minimum, if it fails then an error code can be placed in 'result'
                virtual CLocus *resolveTarget(CMessage *m, HRESULT *result) = NULL;
                                                        // given a message introduced into the system, who gets to run?  'result' matches 'm' and
                                                        // belongs to message's 'm_sender', so these may have to be stored at a later time for valid
                                                        // values, if m->m_sender doesn't get to run right away
                virtual CMessage *resolveMessage(CLocus *l) = NULL;
                                                        // resolveTarget() returned 'l', so now what message does 'l' receive?  Can return NULL
                virtual void restoreContext(CLocus *l, CMessage *m) = NULL;
                                                        // once everything is done, it has been determined that 'l' can now run, being passed 'm'
                                                        // (which may be NULL - which may represent re-execute after preemption or first-start or ...

        public:
                        // ***** interface for inter-broker communication
                virtual HRESULT brokerMessage(const IBroker *broker, CMessage *m)=NULL;
                                                        // 'broker' can't handle 'm' within its own meta-space so it's passing 'm' to another
                                                        // broker - returns whether 'this' can handle the message (that it or one of its children
                                                        // can handle the message) OR 'm' is a IBrokerMsg and 'broker' wishes to communicate
                                                        // directly with this in either case, this message should be handled promptly (i.e. deal with
};                                                      // the message now which may mean recording it, and processing it later when it given the
                                                        // opportunity through the other part of the interface
```

Appendix C: Implementation and Testing Details

C.1 Active Object Implementation

An active object is defined as a combination of functionality found in its class and state information maintained by its locus of execution. There were two options available for actual implementation:

- The object may leverage multiple inheritance (from both the class containing the method and member definitions and a locus class); or
- The two classes may be kept separate and the combination to produce an active object is done conceptually.

Multiple inheritance was not used for one primary reason: not all programming languages support this concept. However, there are other concerns. During migration, an active object's locus of execution object may need replaced by another to be instantiated from a different class. That is, when an object migrates from a broker supporting traditional applications to one that provides soft real-time support, the locus object needs to be replaced with one that provides the necessary extension for soft real-time scheduling information.

More problems may arise if an object is allowed to migrate to a different type of hardware. Certain rules would need to be imposed, such as that the serialized representation of the object has to remain consistent across architectures, and multiple inheritance will only complicate issues.

Given these issues and others, it was decided that the simplest, least prone to error solution is to place synchronization and virtual processor issues into one object: a locus

of execution object. This approach allows the active object's computation class to simply be programmed in a manner that is similar (and in some cases exactly the same) as traditional passive objects. Further, the locus of execution does not necessarily exist in the active object's protection domain to provide a level of security: a malicious object cannot encounter, then modify its protection domain in an attempt gain access to the remaining system.

C.2 Broker Interface Hierarchy Details

Upon close examination of the actual broker class hierarchy presented in Appendix A, one may observe AbstractCPU is not at the base of the broker interface hierarchy, and that AbstractCPU can bind to ApplicationBroker even though ApplicationBroker does not inherit from AbstractCPU. The broker interface is defined as a virtual class with no implementations or member variables. AbstractCPU inherits from and uses this interface but does not provide any implementations and does not define any additional virtual methods. Therefore, AbstractCPU uses the same virtual function table pointer as the broker interface definition. AbstractCPU defines some class variables, such as `currentLocus` and `primaryBroker`, but these do not impact the virtual function table. These variables are defined as class variables to avoid impacting the size of AbstractCPU, which could impact offsets in a derived class. Consequently, no change would have occurred if AbstractCPU had been placed at the bottom of the class hierarchy instead of the broker interface definition. This approach was taken for two reasons:

1. For the average programmer, this model is cleaner. The broker designer does not have to be aware of the AbstractCPU implementation, only that AbstractCPU is present in the system. It also prevents the broker programmer (or compiler) from making invalid assumptions during the design of a broker because assumptions are limited to those of the broker interface.

2. Separating the broker interface from the AbstractCPU implementation allows the Chameleon model to leverage an advantage of the object-oriented paradigm and provide a measure of security. Only the broker interface needs to be publicized and the implementation and any private methods of AbstractCPU can be completely hidden. Although brokers are assumed to be trusted objects, this measure of security is not unreasonable.

C.3 AbstractCPU Details

AbstractCPU only comprises of regular methods that can be compiled to direct (not virtual) function calls, and static data. AbstractCPU performs no dynamic memory allocations to avoid any real-time concerns. It is conceptually created by the system start-up code when the primary broker is instantiated.

C.3.1 Hardware Abstraction Implementation

AbstractCPU provides a complete level of abstraction for all brokers. As stated previously, it performs all basic context switching duties during software exceptions and hardware interrupts. In this respect, it makes full use of the basic locus of execution class. However, memory protection, etc. are left for classes deriving from the basic locus of execution to implement. The assembler code handling memory protection should not be placed in broker code, as such an implementation would directly affect a broker's hardware independence. At this time, memory routines are also defined by AbstractCPU, but are only used by brokers. This design has not impeded flexibility.

C.3.2 Broker Coordinator Algorithm Implementation

The algorithm presented in Figure 4.5 is an abstract representation of the true implementations. This algorithm was actually implemented as two routines: one which handles messages introduced by active objects, and another which handles interrupts

where a change in brokers is expected to execute, for example for preemptive scheduling time-slicing.

The operation of both routines is the same. However, by providing two routines (one where a message is guaranteed to be present), a few processor cycles can be saved for the purposes of performance. The algorithm is simple and straightforward, so this optimization is reasonable given the frequency of invocations.

The number of methods in the broker interface has been minimized to avoid a common pitfall in interface-driven programming paradigms including the object-oriented paradigm. Typically, when a class's interface grows very large, it is exceedingly difficult to correctly and completely convey all the subtleties and semantics to be expected of the interface. Thus, even when an interface is correctly designed and documented, programming errors related to assumptions of the use of the interface to the object are inevitably present. To solve these errors, the implementation of the object often has to be examined, contrary to an important benefit to object-oriented programming. Thus, to avoid these issues, the interface is simply smaller, and doesn't even include methods to notify a broker of when it's meta-space is suspended for other's execution.

C.4 Broker Details

PrimaryBroker implements functionality needed by the primary broker. That is, it must be able to schedule and manage both itself and the objects that it supports in addition to the brokers/meta-spaces it directly supports. It inherits from AbstractCPU and provides implementations for all declared methods. It is instantiated at system start-up. It schedules among the brokers it directly supports using the following algorithm:

- real-time brokers are allowed to pre-allocate a certain percentage of time slices;
- PrimaryBroker then executes any of its own objects;

- finally, PrimaryBroker uses a non-prioritized round-robin scheduling algorithm among the brokers it supports.

PrimaryBroker imposes no assumptions on the utilization of real-time time slices, or for that matter any scheduling performed by other brokers. It schedules among its objects using a prioritized round-robin scheduling algorithm. Although this algorithm is excessively simple, it can be easily be replaced in the future by dynamically replacing PrimaryBroker in favour of another broker object.

ApplicationBroker manages applications. For the purposes of this study, applications are implemented to utilize (and test) communication, memory protection, and exception handling. It inherits directly from the broker interface, and but is supported by the primary broker at run-time. It is designed to utilize service objects known to be managed by the primary broker and cannot support another broker itself. It uses an entirely different (purposely less efficient to illustrate the affects of different schedulers during testing) prioritized round-robin scheduling implementation. It also supports migration to another instance of itself or to another broker. It implements a different communication model from PrimaryBroker too; PrimaryBroker can handle asynchronous messages where ApplicationBroker is designed for only synchronous messaging.

RTBroker manages real-time applications. It inherits from ApplicationBroker and so therefore accepts a similar dictionary of messages (similar API). It schedules any object that it manages that has provided scheduling information. Since it is managed by PrimaryBroker, its guarantee depends on PrimaryBroker's guarantee to it, to execute in known time slices. It supports migration to itself, but not from itself to another broker.

C.4.1 Locus Classes

As shown in Appendix A, PrimaryLocus inherits directly from CLocus (the base locus class). It provides message queues and a scheduling priority for PrimaryBroker management.

ApplicationLocus also inherits from Locus. It is applied to application objects managed by ApplicationBroker. It retains handles to various objects used to optimize performance. It contains a different scheduling priority value as well as information for memory protection and executable code (since that code is not contained in the kernel binary image).

RTLocus inherits from ApplicationLocus and is used by RTBroker. While the object's member variables found in its base class (ApplicationLocus) for memory protection and optimization are still used, the scheduling priority value is unused and real-time information is stored instead of a scheduling priority value.

C.5 Service Object Implementations

Section 7.6 briefly mentions that a service object has to explicitly request to re-enter a wait state pending the arrival of another message. This is an important aspect of the implementation that was chosen to support asynchronous activity. The active service object is allowed to return an immediate success/failure value, then later signal the requesting object upon completion of the request. An example of this is an application object requesting a timed interval: the client object can receive notification that the timer has been set, then notification that the timer has expired. This functionality has application for timeouts for network events, and even simple keyboard monitoring for games. This design is also a primary reason why each Chameleon broker is designed to reschedule during each message pass: not all messages are synchronous, as is assumed for measurement purposes in L4.

C.6 Device Driver Implementations

A limited number of devices are currently supported, and are sufficient for testing purposes. At this time, only console and timer objects exist. Although it is not a requirement, these objects are supported by PrimaryBroker. There is no reason why a device driver object cannot be supported by another broker / meta-space.

One will observe that interrupt handlers are invoked by the hardware abstraction portion of AbstractCPU, and must execute before any of the broker coordinator algorithm. That is, interrupt handlers execute before the context of the currently executing object is completely saved. This design decision was carefully made, and was done for the same performance reasons as in traditional systems. Traditional systems often use this design approach to minimize the overhead of an interrupt handler because interrupts that can or should be ignored occur often. There is no need to incur all of the expense of saving a context (which is variable) for ignored asynchronous events.

Active object portions of device drivers currently disable interrupts to access memory shared with the interrupt handler method. This technique is needed to avoid basic mutual exclusion issues, and is cheaper than introducing synchronization primitives. In addition, synchronization primitives have the potential of blocking the execution of the interrupt handler method, which by design is not allowed by AbstractCPU.

C.6.1 Console Object

The console object manages keyboard input and screen output. It supports up to 16 virtual consoles, so that the user may switch among the running applications. This approach was taken in lieu of building a complex graphical interface that is required for testing purposes. There are basically two parts to the console object: the interrupt handler and the active object portion. The interrupt handler records a keystroke and wakes the

schedulable active object portion via a message to do any further necessary activity. The active object portion can receive requests for keyboard input and console output. All keyboard input requests block until either the request is explicitly cancelled or a keystroke arrives.

C.6.2 Timer Object

The timer object allows other active objects to block (sleep) for a specified duration. It can handle any number of requests, and requests may be cancelled. Currently, it has a granularity of 10 milliseconds (which is the hardware clock interval, also used for preemptive scheduling). It is also comprised of two parts: an interrupt handler counting down any virtual timers, and the active object portion can receive a timer request or cancellation. The active object portion is activated when a timer expires. The timer object cannot provide any real-time guarantees (for receipt of notification of an expired timer) since that policy is defined elsewhere in the operating system – typically by the schedulers.

C.7 RTTI Implementation

RTTI, or *run-time type information* is used extensively. All 'important' objects (i.e., objects that brokers are concerned about to implement the Chameleon model) inherit from a common class: CObject. The compiler's implementation for RTTI was not used to avoid any preconceptions regarding protection domains. Additionally, the compiler was not modified for Chameleon's RTTI implementation. Instead, the source code initializes all RTTI information with a function call that is invoked immediately after the object's constructor method. This implementation of RTTI is far from ideal, but is sufficient for this model. It works for testing purposes, but requires that all classes (that RTTI will be applied to) are explicitly defined by the programmer, and that when this set of classes changes, *all* code (sometimes including applications) must be recompiled.

Improvement of the RTTI implementation, such as the introduction of version control information, is left for future work.

C.8 Chameleon Applications

One primary goal during development of the test-bed operating system was to limit (or prevent) any source-level changes to applications. This goal was achieved in all important respects. While the source code did not change, the source code for the application libraries did to be need modified substantially resulting in the following changes:

- A collection of statically defined RTTI objects are now included in each executable. While this set is a sub-set of those known by kernel-level code (AbstractCPU, service objects, brokers), the set is rather large, comprising of message objects for the API and for interrupts. Functionality to manage the collection of RTTI objects (e.g. linked lists) required introducing more basic classes to the libraries.
- In the original (COSMOS) system, entry via the API to supervisor mode was straightforward: a number representing which system call and a few arguments (possibly packed in structures) were passed via registers. This mechanism was replaced with one much more complicated, where an object is dynamically created on the stack for every system call, and a pointer to the object is passed via a register during the generic software exception. A lot more work is performed for each and every system call, even for simple operations, before the call is even introduced to the system.

C.9 Profiling

Tests in Chameleon had to be profiled. Hardware has proven to be less accurate for timing than expected: the cache is not (presently) controlled in software so is in an

unknown state. Even when the cache is "hot" (loaded with appropriate data), timings still vary, and are attributed to CPU pipelining. For example, tests show that a sequence of NOP's will typically but not always executed in a single CPU cycle.

Bochs is an Intel hardware simulator. A debugger version of Bochs was used for the first step in profiling: breakpoints were set, and executions were followed by the debugger to the completion of the individual test run. The debugger provides op-codes, and disassembles each and every instruction. This information was saved to a file. The state of the simulated CPU and registers, however, was not examined for each instruction. Bochs assumes that each instruction costs exactly one cycle so is useful for instruction counts and dumps but not cycle counts.

The Bochs source is available publicly, so the debugger disassembler code was used as a core for the profiler application. The profiler uses the op-codes to re-disassemble executed code. The cost for each op-code in the CPU instruction set is also publicly available. Using the disassembler code as a basis to estimate cycle costs for each instruction was, therefore, straightforward but could not account for hardware costs and optimizations or situations where instruction costs were variable based on register (parameter) values. The profiler takes a generally pessimistic guess on the cost of an operation.

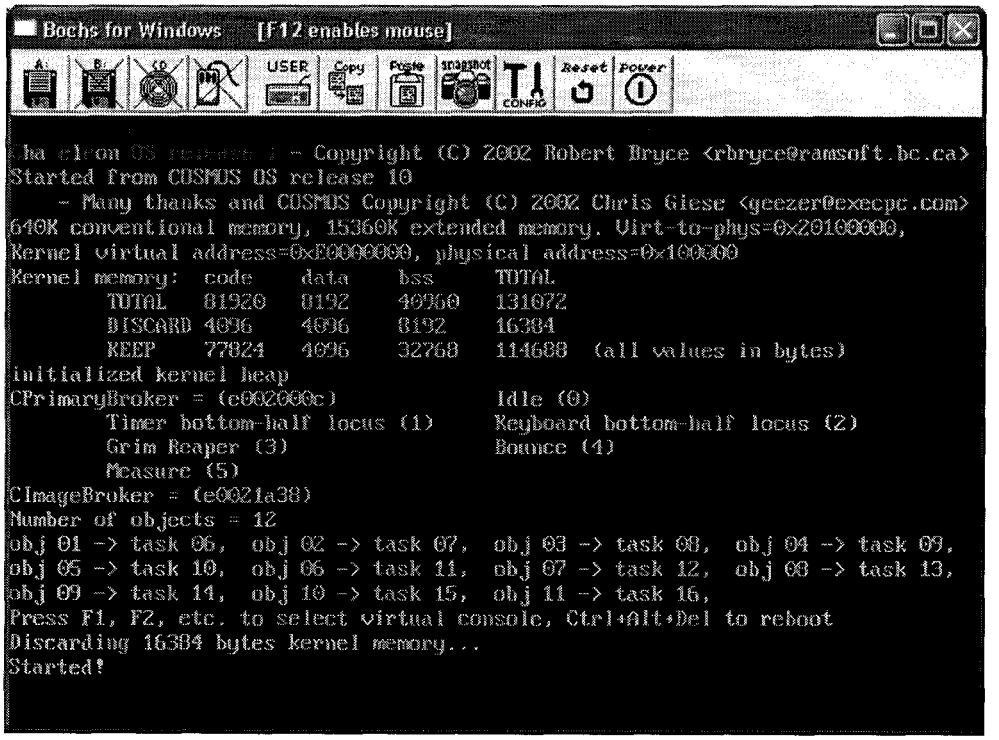
The profiler uses the symbol files produced by the linker to follow function calls. Because applications and system codes are compiled separately, the profiler must accept numerous symbol files.

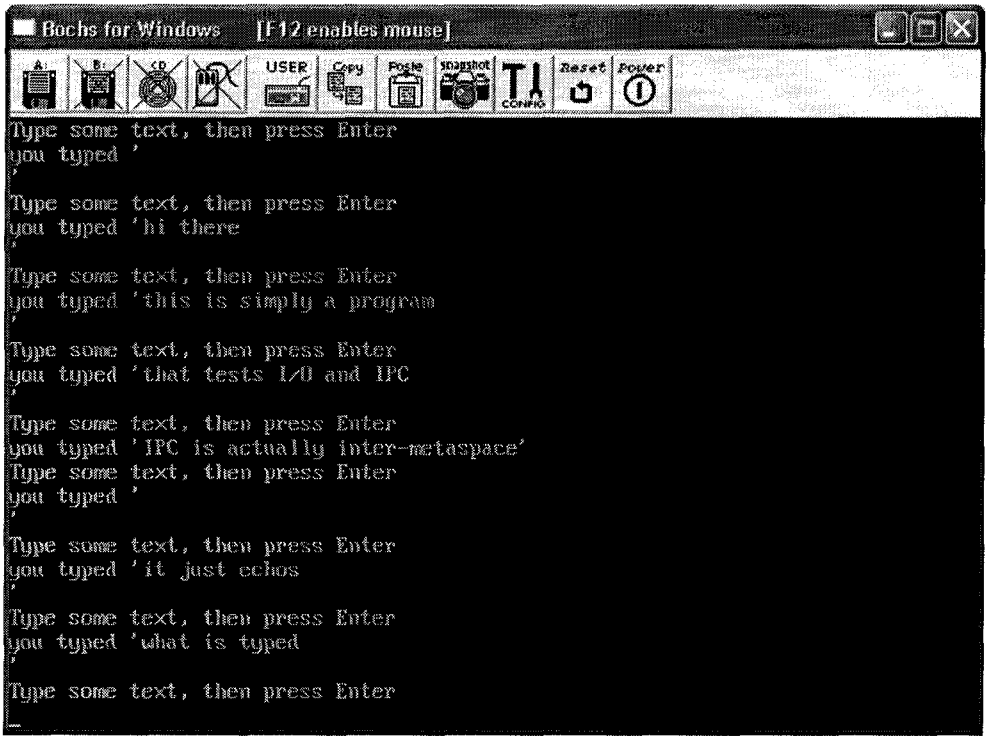
The profiler generates two files for each test: one follows the thread of control through the applications and kernel, and produces total instruction and cycle counts for each function entered. The second file produces an output that provides cycle counts for

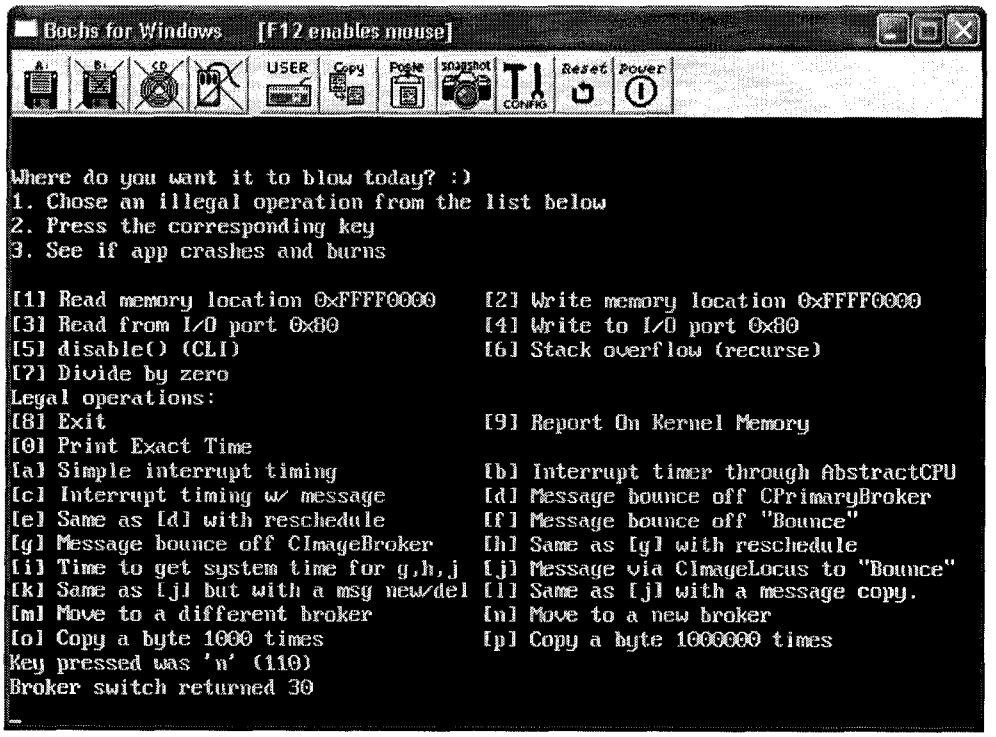
each instruction, for each function call, and total cost (included nested function calls) for each function call. This data was analyzed for the performance chapter.

A variance of roughly 3% has been witnessed in hardware timings. The variance increases slightly with the complexity of the operation(s) being timed. Given that exact instruction sequences are not output by the hardware, it is difficult to measure the effects of each possible variance. The variance in the outputs of the profiler, using data provided by Bochs, has proven to be a lot lower. In addition, the variance is identifiable because exact instruction sequences are available. Variances in the simulator were identified as either unnecessary interrupts occurring, or variations in the execution path through the scheduling algorithm that were only present due to interrupts and disappeared after a couple iterations of the test.

Appendix D: Screen Shots of Chameleon




Bochs for Windows [F12 enables mouse]
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
Type some text, then press Enter
you typed '
'
Type some text, then press Enter
you typed 'hi there
'
Type some text, then press Enter
you typed 'this is simply a program
'
Type some text, then press Enter
you typed 'that tests I/O and IPC
'
Type some text, then press Enter
you typed 'IPC is actually inter-metaspace'
Type some text, then press Enter
you typed '
'
Type some text, then press Enter
you typed 'it just echos
'
Type some text, then press Enter
you typed 'what is typed
'
Type some text, then press Enter


Bochs for Windows [F12 enables mouse]
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
Where do you want it to blow today? :)
1. Chose an illegal operation from the list below
2. Press the corresponding key
3. See if app crashes and burns
[1] Read memory location 0xFFFF0000
[2] Write memory location 0xFFFF0000
[3] Read from I/O port 0x80
[4] Write to I/O port 0x80
[5] disable() (CLI)
[6] Stack overflow (recurse)
[7] Divide by zero
Legal operations:
[8] Exit
[9] Report On Kernel Memory
[0] Print Exact Time
[a] Simple interrupt timing
[b] Interrupt timer through AbstractCPU
[c] Interrupt timing w/ message
[d] Message bounce off CPrimaryBroker
[e] Same as [d] with reschedule
[f] Message bounce off "Bounce"
[g] Message bounce off CImageBroker
[h] Same as [g] with reschedule
[i] Time to get system time for g,h,j
[j] Message via CImageLocus to "Bounce"
[k] Same as [j] but with a msg new/del
[l] Same as [j] with a message copy.
[m] Move to a different broker
[n] Move to a new broker
[o] Copy a byte 1000 times
[p] Copy a byte 1000000 times
Key pressed was 'n' (110)
Broker switch returned 30