Hardware Enforced Protection for Object-Oriented Operating Systems

Preliminary examination paper

David W. Dykstra

December 3, 1990

Abstract

A design for hardware protection between groups of objects is proposed. Objects are grouped into partitioned protection rings. The design provides a transparent C++ interface across protection boundaries. The mechanism used to cross protection boundaries is called the Proxy mechanism. Proxy objects are similar to object capabilities. A Proxy object provides the capability for one partition of one ring to call member functions on an object in any partition of any ring. A co-compiler reads class interface definitions and generates individualized code to help provide the transparent interface. Subclassing across protection boundaries is supported. Classes may be added or replaced without rebooting or recompiling the system. The design will be implemented on the *Choices* operating system which is written in C++.

1 Introduction

An operating system must protect parts of a computer system from other parts. For example, the kernel should be protected from application code, and applications should be protected from each other. Without hardware protection it is impossible to isolate code that has been tested and shown to be reliable from new code that has inadvertent bugs in it, or code that may have even been written by a programmer with intentions to compromise system security.

The C++ model of computation efficiently implements object-oriented programming concepts [RMC90, Jor90]. However, the language does not support hardware protection between objects. The C++ compiler supports compile-time protection between objects, but since all objects are in a single address space with no hardware protection between them, and since the language includes unrestricted pointer manipulation operations, code can easily override the compile-time protection.

1.1 The Problem

The problem that I am trying to solve is this: a generalized hardware-enforced protection mechanism is needed for object-oriented operating systems that includes an interface to an efficient object-oriented language. An efficient object-oriented operating system implementation requires the use of a programming language that efficiently implements object-oriented programming concepts and yet is flexible enough to be used for systems programming. Compiler protection between objects is more efficient than hardware protection between objects but it is not sufficient to protect against the unrestricted pointer operations that are useful in systems programming. Hardware protection boundaries should be set up between groups of objects so objects can be completely protected where necessary and protected by only the compiler when hardware protection is not necessary. A transparent language interface across protection boundaries is needed so a single unified programming model can be used in an entire system. Problems related to providing the transparent interface for an object-oriented language include creating objects across boundaries, passing references to objects across boundaries, and subclassing across boundaries.

There are several reasons why this problem needs a solution:

1. An operating system environment requires hardware-enforced protection between different parts of the system. Programs written by malicious programmers or newly-loaded, untested programs should not be able to corrupt data that the programs do not need complete access to or be able to crash the whole system. Also, an operating system should not rely on a compiler to provide protection since a variety of languages and compilers are available to malicious programmers.
2. A transparent language interface across protection boundaries allows easy movement of portions of code into and out of the kernel; it allows the size of the kernel to be minimized without sacrificing the C++ object-oriented computation model. A smaller kernel has the potential of moving more of the operating system to further-out protection rings, reducing the amount of code that can affect the operation of the entire system. A smaller kernel also allows more of the system to be replaced without taking the system down; this is especially important in systems that require high uptime and is very useful in a research environment.
3. Access between objects that do not need to be protected from each other should not have to pay the cost of hardware protection. The C++ model of computation is efficient and flexible and is a good model to use when protection is not necessary.

1.2 My Solution

This paper proposes a design for hardware-enforced protection between groups of C++ objects. My design includes multiple MULTICS-style "rings of protection" [Org80] where objects in inner rings can access other objects in their own ring and further-out rings. Each ring can be partitioned into multiple address spaces. Accessing objects within a single partition of a ring continues to have the same efficiency that C++ normally has. The same C++ code can call member functions on objects in different rings and on objects in the same ring, but a minimal overhead is incurred to switch

rings. The mechanism that switches protection rings is called the *Proxy* mechanism. Proxy objects are like capabilities [Lev84] that give the right to call member functions on objects in different rings or partitions.

Furthermore, my design provides for dynamic loading of new classes in any partition of any ring without rebooting or recompiling the system. New implementations of classes can also replace existing classes. This kind of dynamic loadability with protection is particularly useful for embedded operating systems in high-uptime applications that need to remain operating during upgrades.

My design will be implemented on the *Choices* [CJR87, CJMR89] operating system, an operating system framework developed at the University of Illinois that is written in C++. The *Choices* kernel is not only written in C++, but it also is designed in an object-oriented manner. Providing a C++ interface to kernel objects outside of the kernel fits more naturally into the *Choices* system than the simple function call (supervisor call) interfaces used in traditional operating systems.

1.3 Comparison to Related Work

Portions of my design are similar to work done elsewhere; the similarities and differences will be explored in this section.

My design uses a combination of a rings of protection model and capabilitites. That combination fits well with providing a transparent C++ interface.

Other systems use a rings of protection model, most notably MULTICS [Org80]. VAX/VMS uses four protection levels with some shared address space [KB84]. The ability to flexibly partition the rings is novel to this work.

Much has been written about capabilities and object capabilities, for example [Lev84] and [BS88]. My design uses a subset of the capabilities model: a Proxy only grants the right to invoke member functions on an object, not other rights such as the ability to read or write an object.

Work has been done in hardware protection of objects in other systems. Several systems use a separate address/protection space for every object, including Clouds [DLA88, PD88], Alpha [ABS89, Nor87], and Eden [ABLN85]. By contrast, my design has a separate protection space for a group of objects, thus avoiding the extra space required to maintain every object and the extra time required to communicate between two objects that trust each other (that is, objects in the same group). A separate address space makes it possible to do fine-grained object migration in a distributed system, but there is no reason why the group of objects in this system could not be migrated together.

Eden uses active objects, where each object has a process associated with it as well as an address space. Communication is done via message passing. My design, like Clouds and Alpha, uses passive objects that are manipulated through member function calls by independent processes.

The Emerald system uses compiler-enforced protection [JLHB88]. The Emerald language does not provide unrestricted pointer operations like C++.

Clouds, Alpha, Eden, and Emerald are all object-based, not object-oriented. In other words, they do not incorporate inheritance and class hierarchies. My design is

for an object-oriented system.

Cross-calls share some design issues with the systems that have an address space per object because cross-calls also need to entirely switch address spaces on member function invocation; these issues include stack-switching and parameter copying. Cross-calls also resemble remote procedure calls (RPC), which also cross address space boundaries as they cross machine boundaries. Examples of systems that support RPC are described in [BN84] and [D+88].

Many other modern operating systems incorporate the minimal kernel concept. Examples include Mach [R+89], V [CZ83], and a kernel for Clouds called Ra [BA+89]. The Ra kernel is written in C++ but it does not provide a C++-like interface to applications.

A system that allows adding code to a running C++ program is described in [DSS90]. That system has several similarities to this one, including the use of first-class classes to keep track of C++ class hierarchies. However, that system does not provide any protection between already running parts and newly loaded parts. It also does not allow for replacing classes with new implementations, only for adding new classes.

1.4 Paper Organization

This paper is organized as follows: Section 2 describes the model of protection. Section 3 discusses how the rings of protection can be implemented on common hardware, and Section 4 describes the Proxy mechanism. Section 5 discusses other topics related to providing a transparent interface. The paper concludes with a look at the state of the implementation of the design.

2 Protection Model

The protection model for this design is a hybrid of protection rings and object capabilities. [Org80] presents a complete description of the protection ring model and [Lev84] presents a description of the object capability model. [BS88] compares the models to each other.

Many operating systems provide protection between different parts of the system by giving each part its own address space. However, pointers to arbitrary data cannot be passed between separate address spaces; all parameters along with the data they reference must be copied to the new address space, and data that is returned must be copied back. To avoid that copying, and to make the interface closely resemble a normal C++ interface where pointers to arbitrary data are commonly passed as parameters, a ring model with shared address spaces is incorporated into the protection model for this design. Each ring can be further partitioned into separate address spaces for cases where sharing address space is not required.

An *object capability* can be viewed as a pointer to an object that also includes the set of operations on the object that the owner of the capability has the right to invoke. Example operations are the ability to read the object directly, to write the object directly, or to execute member functions defined for the objects to indirectly access the object.

A *Proxy* is an object capability that gives the owner the right to invoke member functions on an individual object. The right to read or write objects directly do not come through capabilities in this model; those rights come through the protection ring portion of the model.

2.1 Rings of Protection Model

One partition in one ring is called a *ring-partition*. Let a particular ring-partition be denoted by $r_{p_1 \ldots p_i \ldots p_r}$ where r is the ring number and each p_i is the partition that the ring belongs to in ring i. Let a second ring-partition be denoted by $s_{q_1 \ldots q_i \ldots q_s}$ where s is the ring number and each q_i is the partition that it belongs to in ring i. Ring zero is not partitioned so no partition number is assigned for ring zero.

Figure 1 shows an example with four rings of protection. The number in each ring-partition is the ring number, and the letters represent the partition identifier. The last letter in each identifier distinguishes a ring-partition from other ring-partitions that belong to the same partition of the next inner ring. The other letters identify the ring-partition in the next inner ring. Note that an outer partition never spans more than one partition in the next inner ring. Partitions within a single ring share the same range of virtual addresses, but represent different memory in different address spaces.

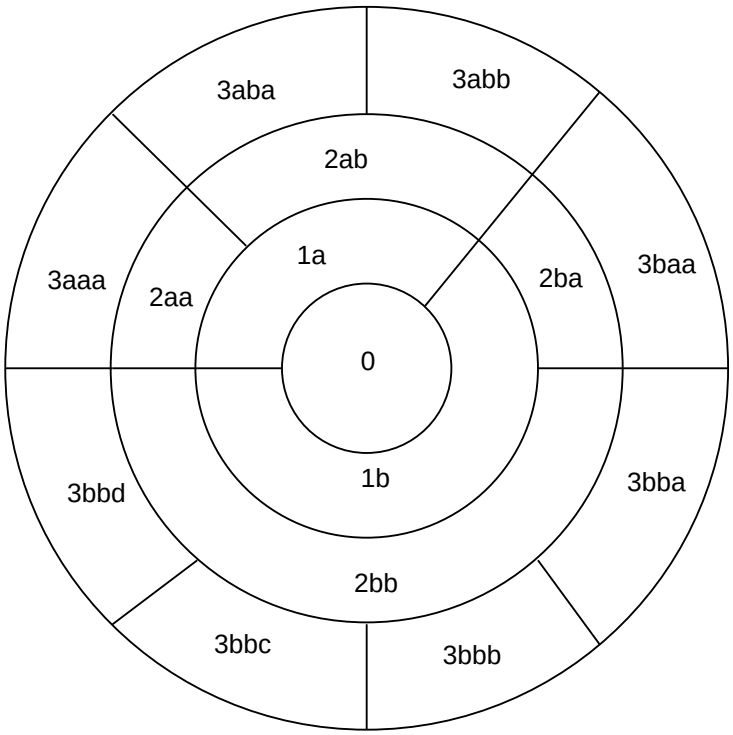

Figure 1: Four Partitioned Rings Example

The term *partition* is used in two ways in this paper: the first is to refer to a portion of a single ring; when used in this way the context will refer to the single ring. The term

partition when used by itself includes all ring-partitions that share address space with a chosen ring-partition. The partition that $r_{p_1 \ldots p_i \ldots p_r}$ is a part of includes all $s_{q_1 \ldots q_i \ldots q_s}$ such that either ($s \leq r$ and $q_i = p_i$ for every $1 \leq i \leq s$) or ($s \geq r$ and $q_i = p_i$ for every $1 \leq i \leq r$). Note that every ring-partition is in the partition of ring-partition zero. In the example, the other ring-partitions in the partition of 2ab are 3aba, 3abb, 1a, and 0.

An address space is represented by a *Domain* in *Choices*[RC89]. An address space contains a ring-partition and all other ring-partitions that are in further-in rings in the same partition. Thus for every ring-partition in a system there is a Domain, and a Domain for ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ refers to the r other Domains for ring-partitions $r - 1_{p_1 \ldots p_i \ldots p_{r-1}}, \ldots, 1_{p_1}, 0$.

A ring-partition has read-write access to all objects in ring-partitions further out in its partition. That is, a ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ has read-write access to objects in all ring-partitions $s_{q_1 \ldots q_i \ldots q_s}$ where $s \geq r$ and $q_i = p_i$ for every $1 \leq i \leq r$. For example, partition 2ab has read-write access to 3aba and 3abb.

Even though a ring-partition has complete read-write access to all objects in ring-partitions further out in its partition, the more trusted ring-partitions will normally follow compiler-enforced protection and only access **public** portions of objects.

2.2 Capability Model

A Proxy (capability) to an object may exist between any two ring-partitions. A Proxy belongs to a single ring-partition and gives the right to invoke member functions on an object in another ring-partition. Invoking a member function on a Proxy causes the protection boundary to be crossed into the object's ring-partition.

Even though a Proxy can exist between any two ring-partitions in the system, only inward calls within a partition can use arbitrary pointer parameter passing semantics. That is, a ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ can call a member function in ring-partition $s_{q_1 \ldots q_i \ldots q_s}$ with pointer passing semantics if $s \leq r$ and $q_i = p_i$ for every $1 \leq i \leq s$. Note that this is the converse of the rule for a ring-partition having read-write access to another ring-partition, because a called ring-partition can accept arbitrary pointers only if it has access to the caller's ring-partition.

Inward calls within a partition are known as *in-calls* and are the most common ring-partition crossings. Outward calls within a partition are known as *out-calls* and calls between two different partitions are *cross-calls*. In those cases pointers to arbitrary data cannot be passed (out-calls actually can with some effort) but these kinds of calls have limited usefulness, as discussed in Section 5.3. A call from ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ to ring-partition $s_{q_1 \ldots q_i \ldots q_s}$ is an *out-call* if $s > r$ and $q_i = p_i$ for every $1 \leq i \leq r$. A call from ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ to ring-partition $s_{q_1 \ldots q_i \ldots q_s}$ is a *cross-call* if neither ($s \leq r$ and $q_i = p_i$ for every $1 \leq i \leq s$) nor ($s > r$ and $q_i = p_i$ for every $1 \leq i \leq r$).

A ring-partition may grant Proxies to any other ring-partition. A ring-partition $r_{p_1 \ldots p_i \ldots p_r}$ may grant Proxies for three kinds of objects:

1. Objects in its own ring-partition.
2. Objects in a ring-partition further-out in its own partition; that, is a ring-partition $s_{q_1 \ldots q_i \ldots q_s}$ such that $s \geq r$ and $q_i = p_i$ for every $1 \leq i \leq r$.

3. Objects in other ring-partitions that it has Proxies for. In other words, granting of Proxies is a transitive operation.

Granting of Proxies occurs through passing pointers to objects or Proxies as parameters to member functions on other Proxies. Thus two ring-partitions in separate partitions cannot grant each other Proxies until one is given a Proxy to the other by a ring-partition that they share. Let $r_{p_1 \ldots p_i \ldots p_r}$ and $s_{q_1 \ldots q_i \ldots q_s}$ be ring-partitions in different partitions. A ring-partition $r'_{p'_1 \ldots p'_i \ldots p'_r}$ that they *share* is defined such that $r' < r$ and $r' < s$ and $p'_i = p_i = q_i$ for every $1 \leq i \leq r'$. For example, ring-partition 1a could take a Proxy to an object in 3aaa and give it to 3abb. From then on 3aaa and 3abb could grant each other Proxies.

3 Multiple Hardware Protection Rings

There are two major ways that most modern computer hardware provides protection: through paging and segmentation.

Paging systems provide virtual address spaces by means of page tables. Page tables contain page table entries. In addition to indicating where in physical memory a page resides, a page table entry indicates protection on each page. Most modern Memory Management Units (MMU's) can indicate whether each page is writable or readable by two protection levels, supervisor and user.

Segmentation systems provide descriptors for each segment of memory. The descriptors indicate where a segment begins in physical memory, how long the segment is, and whether the segment is writable or readable by different protection levels.

The requirements of this design call for multiple hardware protection levels with shared address spaces. The implementation of Proxies calls for portions of the kernel to be readable by all protection levels. Neither a paging system with two protection levels nor a segmentation system directly provides for these requirements.

Some processors such as the 80386 processor provide a combination of segmentation and paging. On these processors, creative use of segments can get around the problem of only two paging protection levels: the address space of the intermediate protection rings are placed above the outermost protection ring in the virtual address space, in reverse order from the outer protection rings through the inner protection rings. All protection rings outside of ring zero, the kernel ring, are run in user mode on the processor but have different segments descriptors associated with each protection ring. All segments still start at virtual address zero so the kernel space can be shared, but the segments are limited in how far up in the address space they can access, thus protecting more privileged data from the outer protection rings. Figure 2 shows an example address space layout for four rings on a system with a combination of segmentation and paging.

Many processors do not provide segmentation but have a page table arrangement called two-level page tables. In a two-level page table system, the first level page table refers to the second level page table and the second level page table refers to the pages. The first level page table is much smaller than the second level page table, typically only one page long.

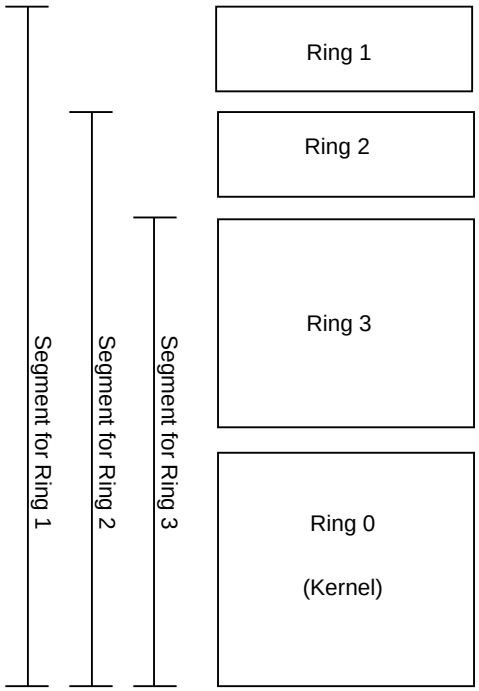

Figure 2: Example Address Space Layout on Segmentation/Paging System

For processors with no segmentation but with two-level page tables, multiple protection rings are achieved by having multiple first-level page tables per Domain. All rings outside of ring zero are run in user mode on the processors but have their own first-level page tables per Domain. For example, if there were four total rings, there would be three first-level page tables per Domain. The first-level page tables all represent the same address space but have different address ranges accessible to user-mode code. Loading a new page table at the time of the protection ring switch forces the Translation Lookaside Buffer caches to be flushed. However, preliminary measurements indicate that doing so will not add much more time to a Proxy call because the processors on which *Choices* runs only have small TLB caches.

Figure 3 shows an example of different views of the same address space on a system with two-level page tables and two hardware protection levels. The two protection levels are called "Kernel" and "User". The example shows four protection rings which requires three different views; ring zero can use any of the views because it operates in kernel mode. Each view corresponds to a first-level page table. Each first-level page table is divided into four areas corresponding to the four rings. Each ring has read/write access to its own space and all further-out rings. Ring zero shows a portion that is read only to all rings because the implementation of Proxies requires a portion of address space that is readable by all rings but writable only by ring zero.

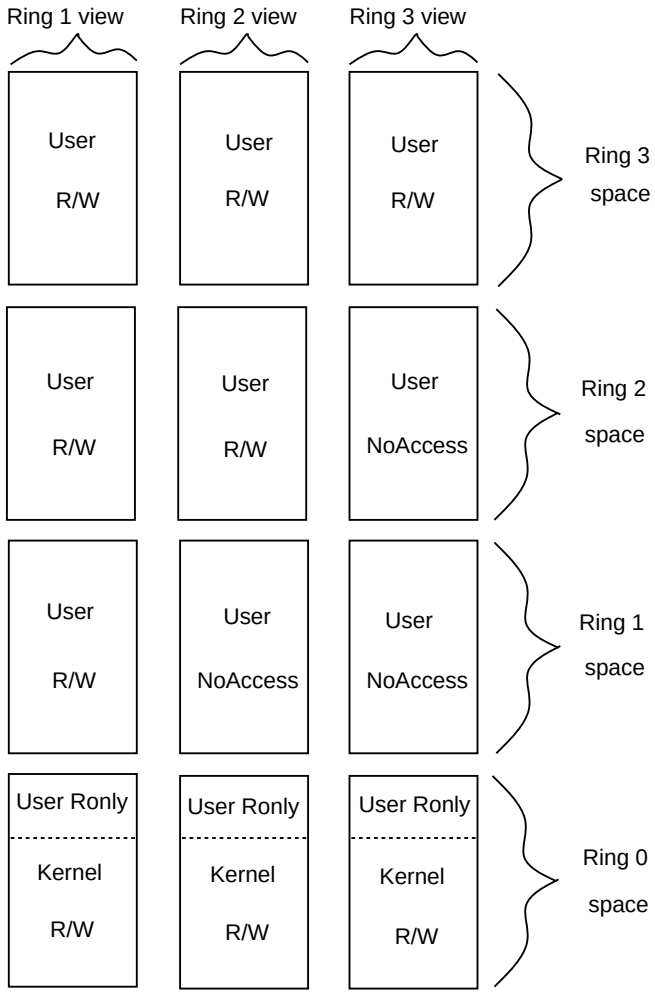

Figure 3: Example Two-level Page Table Address Space Views

4 Proxy Objects

The mechanism used to cross protection boundaries is called the Proxy mechanism [Rus90]. An object is represented outside of its protection ring with another object called a Proxy object. A Proxy object contains a pointer to the "real" or "proxied" object and transparently behaves as if it were the proxied object. Proxied objects are instances of a subclass of a class called ProxiableObject. The term *ProxiableObject* is used in this paper to refer to objects that can have a Proxy refer to them.

Proxy objects also contain a pointer to the Domain that the proxied object is in for use in cross-calls and out-calls. Each individual Proxy object can only be used by a single ring-partition and is associated with the Domain that represents the ring-partition. Thus a Proxy object is a one-to-one mapping from a ring-partition to a ProxiableObject in a different ring-partition.

Proxy objects are controlled by the kernel; they are writable only by the innermost

ring but are readable by the ring-partition that uses them.

Proxy objects take advantage of C++ virtual function tables. Each virtual function in the Proxy class is actually a stub that traps to the kernel. The kernel trap handler verifies that the 'this' pointer, which points to the Proxy object, is a valid Proxy pointer and translates the Proxy pointer to refer to the proxied object instead. The trap handler then switches protection to the new ring-partition and invokes the real member function on the proxied object, taking care that the return address of the function is set so the ring-partition can be restored when the function returns.

Along with the switch of ring-partitions is a switch to a different stack; stack-switching is discussed further in Section 5.4.

4.1 Wrapper Functions

The real member functions on proxied objects are not called directly, they are called via wrapper functions that are automatically generated. The wrapper functions provide an important part of the transparent interface. A co-compiler called *Proxify*++ generates wrapper functions for every member function that can be invoked through a Proxy. This co-compiler reads class definitions from header files and uses them to generate C++ code for wrapper functions.

Wrapper functions serve several purposes as listed below. This section only covers the most common case, in-calls. Out-calls and cross-calls are handled differently as discussed in Section 5.3.

1. The wrapper functions copy the correct number of parameters from the outer ring's stack to the inner ring's stack. This avoids having to copy a fixed number of parameters for all member functions. Not all compilers/processors pass all parameters on the stack, but since the wrapper functions are automatically generated it is easy to have machine dependent versions.

2. If the member function returns a pointer to a ProxiableObject, the member function's wrapper function allocates a Proxy to that object and returns that instead. In this way Proxies get allocated only when needed: when a member function is invoked by a Proxy, a Proxy will be returned, and when the member function is invoked directly, the object itself will be returned.

3. The wrapper functions handle validation of parameters that are pointers to ProxiableObjects. This is the most important purpose for the wrapper functions; the other things that wrapper functions do could be handled (although not as well) without individualized wrapper functions, but validation of parameters requires knowledge of the individual member functions that are being called. The primary motivating factor in doing this validation is that it is assumed that passing a pointer to a ProxiableObject implies that member functions will eventually be invoked on that object. If those member functions are to execute in a further-in ring than the caller or another partition than the caller, the data of that object must be protected from corruption by the caller. Validation of each pointer to a ProxiableObject is divided into several steps as listed below.

(a) If the pointer references an object in the caller's space and the type of the ProxiableObject is one that may be subclassed outside of the ring-partition (see Section 5.2), a Proxy to that object is allocated and passed to the member function in place of the pointer to the object. There is no need to validate the object any further because if anything is wrong with it, out-calls to member functions on that object will only corrupt things in the caller's space. If the parameter is of a type which is not to be subclassed outside of the ring-partition, it is an error.

(b) If the pointer points outside of the caller's space and does not reference a valid Proxy, it is an error. The only access that the caller has to objects outside of its space is through Proxies.

(c) If the pointer refers to a valid Proxy to an object in a further-in ring than the caller or an object outside the caller's partition, the type of the object must be validated. If the type of the object were incorrect, member functions called on that object would not be the ones the caller intends. For example, a call to member function number nine on an object that is expected to be type A would actually call a totally different member function number nine on an object of type B; among other things, the parameters will be incorrect which can cause corruption of data.
The type of an object is determined by asking the object what its type is. The object can be trusted because the Proxy to it has already been validated. Because of the hierarchy of types and inheritance in C++, any subtype of the required type is valid. The type hierarchy is mirrored by a run-time representation of the classes called first-class classes [MCK91, IL90]. First-class classes are discussed further in Section 5.2.

(d) If the pointer refers to a Proxy to an object in a further-out ring in the caller's partition, no type validation is necessary. If the type is incorrect, only the further-out ring could get corrupted and the caller has the right to do that. In addition, the answer that an object in a further-out ring gives for its type cannot be trusted.

(e) If the pointer refers to a Proxy to an object in the callee's ring-partition, and the type of the object is correct, the Proxy is stripped off and the real member function is given a pointer to the proxied object. There are two reasons for this. The first is efficiency; there is no need for the overhead of the Proxy traps within the same ring-partition. The second reason is that if the Proxy were left in then non-virtual member functions and inline member functions, which code in the same ring-partition may use, would not go through the Proxy trap and thus would not get the "this" pointer translated to the real object. An incorrect "this" pointer will cause the functions to treat the memory of the Proxy object as if it were the real object and cause data corruption.

Proxify++ also generates for each class a table that points to the wrapper functions for the class. At the time a Proxy is allocated, this table is associated with the Proxy so the table can be quickly found at the time of a Proxy trap.

4.2 Selecting Proxiable Member Functions

Not all member functions on a ProxiableObject should be callable through Proxies; there are some member functions that a programmer may want to be used within a program but may not want to be callable from outside the ring-partition. This especially applies to those member functions that are declared to be **protected**. Even though the compiler protects them, a malicious programmer can write code to call them.

The approach taken in this design is to add a new keyword **proxiable** to the declaration of member functions that are to be callable through a Proxy. The C++ compiler itself does not change; at compile time, the **proxiable** keyword is removed by the preprocessor. The *Proxify*++ co-compiler reads the declaration of classes looking for the special keyword and only generates wrapper functions for those member functions that are declared to be proxiable. Attempts to call non-proxiable member functions generate errors. Requiring the programmer to use a keyword on member functions that are to be proxiable rather than somehow indicating which member functions are not to be proxiable reduces the chances that a programmer will open a security hole by inadvertently allowing a dangerous member function to be called.

Proxify++ also generates alternate class-description header files for other programs to include. These header files define only the interface to the class. This has several advantages over requiring other programs to include the original header files:

1. Non-proxiable member functions are removed from the class description so attempts to call them can be caught at compile time.
2. Since only the interface is needed outside the program, there is no need for other programs to be recompiled when there are changes to the details of the implementation.
3. With the details of the implementation removed, fewer nested header files need be included because header files are often included to support the implementation of the object only.
4. All proxiable member functions in the new header files are declared with the keyword **virtual** because the virtual function mechanism is used by Proxies. The real member functions need not be virtual.

4.3 Reference Counting

To help remove objects no longer in use, all ProxiableObjects in *Choices* are reference counted. Thus objects can be shared between different pieces of the system and are only deleted when their reference count becomes zero. Calls to the member functions that implement reference counting are caught and handled specially by the Proxy trap handler.

When a Proxy is allocated, the reference count on the proxied object is incremented once. From then on, when the *reference* member function on the Proxy is called, the reference count on the Proxy itself is incremented rather than passing the call through to the proxied object. Similarly, the *unreference* member function decrements the

reference count on the Proxy only. When the reference count on the Proxy becomes zero, the Proxy is deleted and the *unreference* member function on the proxied object is called once via the normal proxy trap mechanism.

Trapping the reference counting calls on Proxies in this way has two purposes: first, it prevents a program from corrupting the reference count on the proxied object by incrementing or decrementing the reference count further than the program is authorized to do. Second, it provides a convenient method of deallocating Proxies when they are no longer needed.

5 Other Transparent-Interface Topics

This section discusses several topics related to providing a transparent C++ interface across protection rings boundaries: obtaining new Proxies, subclassing across protection boundaries, out-calls and cross-calls, switching stacks, and dynamic addition and replacement of classes.

5.1 Obtaining Proxies

Programs can obtain Proxies for both pre-existing objects and for new objects. Every object that a program has a Proxy to has the potential of returning more Proxies to pre-existing objects or new objects, but there are some special objects that are provided for this purpose.

Every program starts with one Proxy to a kernel object called TheSystemInterface. One of the member functions on TheSystemInterface returns a Proxy to the StandardNameServer object for the program. A pre-existing object that is to be made accessible to the program is given a name and *bound* into the StandardNameServer. When the program does a *lookup* on the StandardNameServer with that name, a Proxy to the object is automatically allocated and returned. There is a hierarchy of NameServers that follows the hierarchy of ring-partitions; if a name is not found in the first NameServer, it is looked up in the next inner ring's NameServer, and so on until the name is found or until it is not found in the innermost ring.

Another member function on TheSystemInterface called *constructProxy* is used to call proxiable constructors and to return Proxies to them. Proxiable constructors are marked with the **proxiable** keyword in their declaration and processed by *Proxify*++. Constructors that are proxiable can be called by using the C++ keyword *new* using the same syntax for creating objects within a program. This works by going through stubs that are generated by *Proxify*++. The stubs call the *constructProxy* function with a string name of the constructor and the rest of the parameters for the constructor. The *constructProxy* function then looks up the constructor name in a hierarchy of ConstructorNameServers similar to the hierarchy of object name servers. *Proxify*++ automatically puts in the string names in the ConstructorNameServers from the same list of **proxiable** constructors that is used to generate the stubs. After the constructor is looked up in the ConstructorNameServer, *constructProxy* calls the constructor and returns a Proxy to the constructed object.

Since the constructor stubs are generated at compile time, a program can only use **new** on class types that are known at compile time. In general, this is not a problem. Another approach as taken in [DSS90] and [MCK91] is to add a member function to the first-class class objects to create a new object of that class. The problem with that is that parameters can not easily be passed to the constructors because different constructors take different kinds of parameters, and another *init* function must be provided to initialize the objects after they are created. Constructor stubs make it easier to move code out of one program into another without having to modify the way objects are created.

5.2 Subclassing Across Rings

Subclassing a class defined in a different ring-partition is supported. In C++, an object of a subclass can be used wherever an object of a parent class is expected. Any individual **virtual** member function of the parent class can be overridden by a member function in the child class. Member functions that are not overridden are *inherited* from the parent class.

Space for an object is allocated within one ring-partition. The object's member functions will also be stored in that ring-partition and must execute there. If an object belonged to ring one, for example, but a member function was invoked on the object in ring zero, the ring zero member function could not be sure that ring one code had not corrupted data in the object unless the ring zero member function validated the data at the beginning of every call. This would not be a problem if the ring zero member function only manipulated the data in the object, but it is quite likely that there could be side effects to other ring zero data based on what is in the object.

For this reason, the code for parent classes are copied into programs that define subclasses. Another possibility might have been to allow further-out rings to have read-only access to further-in rings and execute code directly out of them, but that does not work because the further-in ring code may attempt to write to global variables that are not accessible to the further-out ring.

It is possible to share the parent class code between different programs loaded into the same ring-partition, if dynamic linking is used, but that would place an unnecessary dependency between the programs such that they could not be independently replaced. That is, the program that contained the parent class code would need to remain in place so other programs could use the parent class code, even if subclasses in the program were no longer in use.

Copying the code for the parent classes also fits better with the design of Proxies since there is one ring-partition associated with an entire object, not one for every member function on an object.

Not all classes should be allowed to be subclassed outside the program in which they are first defined, for several reasons:

1. Member function calls on instances of these classes can not be trusted since they run in a less privileged ring.
2. Since the subclasses can be defined in a further-out ring than the parent classes, their member functions are likely to be called as out-calls. Out-calls do not

have the ability to pass pointers to arbitrary data as parameters so the member functions must take that into account.

3. Since the subclasses will be accessed through a Proxy, the parent class must be restricted to have no non-virtual or inline member functions. Even **protected** and **private** member functions must be restricted. The reason for the restriction is that non-virtual or inline member functions are not trapped by a Proxy and thus will not switch to the proper ring-partition.

Classes that may be subclassed outside their ring-partitions are marked with the **proxiable** keyword in the definition of the entire class. *Proxify*++ enforces the restriction of no non-virtual or inline member functions on classes marked **proxiable**. It also requires that every member function in the class be marked individually with the **proxiable** keyword to emphasize that each member function can be replaced by a proxied member function.

Proxify++ takes the proxiable attribute of the class into account when generating wrapper functions by only allowing parameters of Proxies to objects outside of the ring-partition if the type of the parameter is for a class that was marked **proxiable**.

As mentioned earlier, the hierarchy of classes are mirrored by a hierarchy of objects called first-class classes. Each object in the hierarchy is an instance of the class called Class. Each time a new program is initialized, the subclasses of ProxiableObject defined in the program are added to the Class object hierarchy. The Class instances are also added to the NameServer for the ring-partition that the program is loaded into so they can be easily retrieved. Replacing classes that already exist is discussed further in Section 5.5.

5.3 Out-calls and Cross-calls

Out-calls and cross-calls are more difficult to implement than in-calls because the called member functions cannot be trusted. Also, the called member functions do not have access to the caller's space, including the caller's stack. Out-calls and cross-calls are slower than in-calls, but they are less common.

Despite the restricted nature of out-calls and cross-calls and the difficulty of supporting them, there are some important uses for them. An example use for out-calls is exception handling: when inner ring code encounters some kinds of recoverable error conditions, it could invoke a member function provided by outer ring code to try and recover from the error. An example use for cross-calls is to provide functionality similar to signals in UNIX. Another use for cross-calls is to move a program that does not need to be in an inner ring out to an outer ring.

Much of the difficulty of supporting out-calls and cross-calls is with parameter passing. MULTICS does not support passing parameters on out-calls at all but this design includes it.

Because the called member functions cannot be trusted, the functionality that is provided by wrapper functions on in-calls cannot be done by the callee's ring-partition. The callee's ring-partition does not have access to the caller's stack so it cannot copy parameters from there. It also does not have the right to allocate Proxies back to

ProxiableObjects that the caller may pass in as parameters. For this reason, the caller on out-calls and cross-calls provides a table that indicates how many parameters each member function has, which parameters are ProxiableObjects, and the return type of each member function that returns a ProxiableObject. This table is generated by *Proxify*++ and associated with the Proxy at the time the Proxy is allocated. The Proxy trap handler takes care of using that table, allocating Proxies when necessary, and checking the return type if necessary.

Cross-calls have the added complication that neither the caller nor the callee can trust the other. In addition to the Proxy allocation done by the caller as on out-calls, Proxy checking and type checking that are normally done by wrapper functions on in-calls must be done by the callee. For that purpose, *Proxify*++ generates yet another table for use by the callee with information about which parameters of each member function are ProxiableObjects, the type of each ProxiableObject parameter, and whether each member function returns a ProxiableObject. Cross-call Proxies have both this table and the table mentioned in the previous paragraph associated with them; the Proxy trap handler uses the tables.

The same table that is generated for use in cross-calls can probably be used for in-calls instead of wrapper functions. This possibility will be investigated and the performance will be compared to wrapper functions.

5.4 Switching Stacks

Any time a ring-partition is crossed, a stack switch must occur. The need for a stack switch on out-calls and cross-calls is obvious, but the need for a switch on an in-call is more subtle. For example, a process that makes an in-call from ring one to ring zero has the right to access the stack in ring one, so one might think that a stack switch would be unnecessary. The problem is that if a stack switch does not occur then other processes executing in the same partition of ring one could corrupt the stack of the process that is now running in ring zero.

Thus for in-calls alone each process requires a stack allocated from the ring-partition in which it is created and each ring further in. To avoid allocating stacks that are never used, the inner stacks are allocated the first time a Proxy call requires them rather than at process creation time. After that the stacks are reserved until the process dies so in-calls can be as fast as possible. The number of stacks that can be reserved for each process is small, limited by the number of rings.

For out-calls and cross-calls, however, stacks cannot be reserved indefinitely as on in-calls because there is no way of knowing how many different ring-partitions will be called by each process. A small cache of stacks could be kept for each process to reduce the time for repeated out-calls or cross-calls to the same ring-partition, or the stacks could be allocated when needed and immediately freed when the calls are complete.

5.5 Adding and Replacing Classes

Implementations of related classes are expected to be grouped together into *load modules*; that is, separately compiled and linked programs. Load modules can be dynami-

cally loaded into any existing ring-partition at any time. A new ring-partition can also be created in any ring (other than ring zero) prior to loading a load module into it. A relocating loader loads the modules so they can be relocated to any address in any ring.

Protection against unprivileged processes invoking the loader is beyond the scope of this project, but loading load modules outside of the ring-partition of the invoker should be restricted to a super-user process or some equivalent.

The initialization code in a newly loaded load module creates a new Class object for each proxiable class supported in the load module and adds each of the Class objects to the NameServer for that ring-partition. If the new load module replaces earlier versions of the same classes in the same ring-partition, the old first-class class objects representing the classes in the old load module will be replaced in the NameServer by the new ones. If classes are replaced in a further-out ring than the original definition, the original definition is not removed from its own NameServer but the new definition "hides" the original from further-out rings.

Load modules themselves are reference counted. A load module is deleted when none of its classes remain in the NameServer, there are no more instances of any its classes, and there are no more processes executing code in it.

The current design calls for no Proxies between two different load modules in the same ring-partition because no protection boundary needs to be crossed during a call. This will be investigated further, however, because Proxy objects could be a convenient place to put aids for replacing load modules if Proxies were always used when objects are referenced outside of a load module. One advantage is that the test for no instances of a class could be replaced with a test for no in-pointing Proxy objects. Another advantage is that the deletion of an obsolete load module could be sped up if all the Proxies pointing into a load module were kept in a list to allow an operation to kill or otherwise notify all processes that are using classes in the old load module.

To help ensure that programs use the correct interface to ProxiableObjects, some kind of version control would be useful. A version number could be automatically inserted into the definition of the class interface by *Proxify*++ whenever the interfaces changes. The version number that a load module requires for each class could be registered at load module initialization time and checked whenever a Proxy is allocated. Version control will be investigated further.

6 Conclusion

In conclusion, I will cover the current state of the implementation of my design and propose work that I plan to do to demonstrate the thesis of this paper.

6.1 Completed components

The following components of my design are operational:

1. Basic Proxies for in-calls are operational, including multiple rings on the 80386 processor. An initial simple implementation of Proxies for user to kernel was

already in place before I began work on it, but I completely reimplemented it and made many improvements.

2. Proxies are reference counted and the reference counting member functions are trapped and applied to the Proxies instead of the proxied objects, as described earlier in this paper.
3. Proxies are allocated out of one large pool that is in shared kernel memory using a class called the ProxyPoolManager. This is necessary at this time because Proxies to kernel objects are sometimes used within the kernel even when the Domain that owns the Proxies is not active. When wrapper functions are automatically generated, load modules will not be using Proxies to objects in their own modules so the Proxies themselves will be able to be allocated in the address space of the Domains that own them.
4. Proxiable objects can be created by using **new** through constructor stubs as described earlier in this paper.

6.2 Components to be completed

I propose to do the following:

1. Write the *Proxify*++ co-compiler. Currently some of the functionality that will be provided by *Proxify*++ is provided through hand-written wrapper classes. *Proxify*++ will eliminate the need for the hand-written classes and will make the Proxy interface transparent.
2. Implement the partitioning of inner rings. Currently only the outermost ring can be partitioned.
3. Implement protection between outer rings. The current prototype has the rings other than the outermost rings in shared space but only the innermost ring is protected from other rings.
4. Implement out-calls and cross-calls. Only in-calls are implemented at this time.
5. Implement stack switching on all ring-partition crossings. The prototype only switches stacks on crossings into and out of the innermost ring.
6. Augment first-class classes, NameServers, and ConstructorNameServers to follow the hierarchy of ring-partitions.
7. Make use of a relocating loader (written by others) to load programs at any address. The prototype loads at fixed addresses.
8. Take measurements to evaluate the performance of all ring-partition crossings with and without passing ProxiableObjects as parameters.
9. Make everything operational on at least two different computers, the AT&T 6386 PC and the Encore Multimax.

6.3 Things I do not plan to do

The following are things I do not plan to do that someone may want to investigate in the future:

1. Add support for multiple inheritance. Support for multiple inheritance is not in the current plan, because the *Choices* researchers have chosen to use only single inheritance and because C++ currently does not have an efficient implementation of multiple inheritance.
2. Add more control over the security of granting Proxies by use of discretionary access control. An example of such work is found in [BTMD89].
3. Investigate the applicability of this work to distributed operating systems and migration of ring-partitions across machine boundaries.

6.4 Related work I have done

I have worked on several related things during my graduate study that have helped prepare me for doing this research:

1. I implemented timeout timers and periodic timers in *Choices*. It was apparent from the beginning that the object-oriented interface to these timers should be accessible outside of the kernel, and that a C++ user-kernel interface was necessary.
2. I investigated making a network switch controller (Datakit) based on *Choices* and constructed a prototype. The need for on-line upgradability and a C++ interface across protection boundaries in that controller has been a major motivating factor for this work.
3. I ported the *Choices* operating system to the AT&T 6386 PC. This was the first complete port of *Choices* and it helped make *Choices* portable. The porting work helped me to understand the internals of *Choices* and to see that it was lacking in this area.

7 Acknowledgements

The original implementation of Proxies as well as much of the implementation of the *Choices* kernel was done by Vince Russo. Peter Madany introduced first-class classes. Peter also helped greatly with suggestions for revisions to this paper. Roy Campbell is my research advisor and helped me to define this topic.

References

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–58, January 1985.

[ABS89] H. J. Antonisse, J. W. Benoit, and B. G. Silverman. Alpha: a new generation real-time decentralized operating system. In *Proceedings of the Annual AI Systems in Government Conference*, pages 315–318. IEEE, March 1989.

[BA+89] J. M. Bernabeu-Auban et al. The architecture of Ra: a kernel for Clouds. In *Proceedings of the Twenty-Second Annual Hawaii International Converence on System Sciences*, pages 936–945 vol. 2. IEEE, January 1989.

[BN84] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, II(1):39–59, February 1984.

[BS88] L. Bic and A. C. Shaw. *The Logical Design of Operating Systems*, chapter 7. Prentice Hall, Englewood Cliffs, NJ, 1988.

[BTMD89] M. Branstad, H. Tajalli, F. Mayer, and D. Dalva. Access meditation in a message passing kernel. In *Proceedings 1989 IEEE Symposium on Security and Privacy*, pages 66–72, Oakland, California, May 1989.

[CJMR89] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[CJR87] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3), July 1987.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128–140, October 1983.

[D+88] T. H. Dineen et al. The Network Computing Architecture and system: an environment for developing distributed applications. In *Digest of Papers: COMPCON Spring 88*, pages 296–299, San Francisco, California, 1988.

[DLA88] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The Clouds Distributed Operating System: functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, June 1988.

[DSS90] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[IL90] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Jor90] D. Jordan. Implementation Benefits of C++ Language Mechanisms. *Communications of the ACM*, 33(9):61–64, September 1990.

[KB84] L. J. Kenah and S. F. Bate. *VAX/VMS Internals and Data Structures.* Digital Press, Bedford, MA, 1984.

[Lev84] H. M. Levy. *Capability based Computer Systems.* Digital Press, Bedford, MA, 1984.

[MCK91] Peter W. Madany, Roy H. Campbell, and Panogiotis Kougiouris. Experiences Building an Object-Oriented System in C++. In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991. Submitted for publication.

[Nor87] J. D. Northcutt. Mechanisms for Reliable Distributed Real-Time Operating Systems. *Perspectives in Computing*, 21, 1987.

[Org80] E. I. Organick. *The Multics System*, chapter 4. MIT Press, Cambridge, MA, 1980.

[PD88] D. V. Pitts and P. Dasgupta. Object Memory and Storage Management in the Clouds Kernel. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17. IEEE, June 1988.

[R+89] R. Rashid et al. Mach: a system software kernel. In *Digest of Papers: COMPCON Spring 89*, pages 176–178, San Francisco, California, 1989.

[RC89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[RMC90] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[Rus90] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.