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.