An Operating System Structure for Wide-Address Architectures

Jeffrey S. Chase

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 95-08-06
August, 1995

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer.

An Operating System Structure for Wide-Address Architectures

by

Jeffrey Scott Chase

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1995

Approved by ____
Chairperson of Supervisory Committee

____
Chairperson of Supervisory Committee

____

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

An Operating System Structure for Wide-Address Architectures

by Jeffrey Scott Chase

Chairpersons of the Supervisory Committee: Professor Henry M. Levy
Professor Edward D. Lazowska
Department of Computer Science
and Engineering

This dissertation develops and explores a new operating system model made possible by wide-address architectures (e.g., 64-bit processors such as Digital's Alpha family and the MIPS R4000) that are now appearing on the market. The very large virtual address spaces supported by these architectures have undermined a basic design principle of 32-bit operating systems: the assumption that virtual address space is a scarce resource that must be conserved by redefining address mappings in each active process. This presents an opportunity for a *single virtual address space operating system* with a global virtual-physical mapping for all programs. The single address space approach has several attractive properties; for example, it naturally supports cooperation among application components using uniformly addressed shared memory and memory-mapped long-term storage.

The contributions of this dissertation are to: (1) define abstractions and primitives for a single address space operating system called Opal, (2) implement and evaluate a prototype of the Opal system, (3) experiment with new application structures made possible by the Opal model, and (4) explore tradeoffs of the single address space approach for the operating system and its applications. The Opal design presents novel solutions to issues raised by the single address space approach, including address space management, protected data management, and resource reclamation.

This dissertation reaches three primary conclusions. First, the Opal model can be implemented on wide-address paged RISC architectures using standard microkernels, with performance comparable to other operating system environments supported by those platforms. Second, the model expands the structuring choices available to applications, including new options that are useful to important classes of applications, while imposing acceptable costs on applications that do not directly benefit from its features. Third, the model attains the key goals of modular sharing and protection that were common to many earlier systems, while preserving the simplicity, generality and performance responsible for the success of current systems based on private virtual address spaces.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First, I would like to thank my advisors, Hank Levy and Ed Lazowska. Both have helped me in more ways than I can mention, but I especially thank Hank for his vision and Ed for his expertly delivered criticism and encouragement. Both are well-deserving of their excellent reputations, and I am most grateful for their faith and patience during my tenure as a graduate student.

In addition, Washington is blessed with excellent staff support. Special thanks go to Jan Sanislo, always the man behind the curtain, and Frankye Jones, who regularly saves graduate students from themselves.

Many students have been involved with the Opal project and have contributed to the research described in this dissertation. At the top of the list is Mike Feeley, who worked in the trenches throughout the project, and without whom Opal would not have been possible. Other collaborators include Ted Romer, Vivek Narasayya, Curtis Brown, Jay (Thierry) Han, Dylan McNamee, Simon Auyeung, Bik Kwan Winnie Ng, Miche Baker-Harvey, Jeff Kaufmann, Rene Schmidt, and Ashutosh Tiwary. I would also like to thank John Wilkes and Sandy Kaplan for the contributions they offered purely from generosity.

I am indebted to Digital Equipment Corporation, Intel Corporation, and the Boeing Company for supporting me and my research. Among the people who have worked with me and supported me at these companies, I would especially like to thank Reid Brown, Kent Ferson, Fred Glover, Roger Heinen, and Chet Juszczak at Digital, Kevin Kahn at Intel, and Bob Abarbanel and Ashutosh Tiwary at Boeing. This work was also supported by the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and the Washington Technology Center.

Finally, I would like to thank my father, who showed me this path, my mother who helped me over more than a few obstacles along the way, and my wife Nonna, who is waiting for me at the end.

Chapter 1

Introduction

> *"Meta-instructions"...embody powers mostly absent from contemporary programming languages, but essential to the implementation of computation processes in a multiprogrammed computer system. These powers relate to (1) parallel processing, (2) naming objects of computation, and (3) protection of computing entities from unauthorized access.*
>
> *-- Dennis and Van Horn, 1966*

Every operating system defines a set of basic abstractions and relationships between them, including primitives for combining the abstractions to structure application software. An early paper by Dennis and Van Horn [DV66] refers to these abstractions and primitives as "meta-instructions". The meta-instructions comprise a *system model* or "language" used by application programmers to express how their programs interact and make use of the functions provided by the underlying computer hardware. The degree to which this model matches programmer needs significantly affects application development costs, application quality, and overall system performance.

1.1 Addressing and Protection

This dissertation concerns the relationship between two fundamental elements of any operating system model: memory *addressing* and memory *protection*. An addressing scheme defines how instructions executing in the processor name data operands in memory, and how references are passed among programs that share data in common. A protection scheme allows programmers to restrict the allowable references by each software entity at each point in time. Addressing and protection facilities are defined by hardware and system software, and are closely related.

As software systems grow in size and complexity, it becomes more important to restrict access to sensitive information and to limit the damage from accidental or malicious software misbehavior. The system must provide protection in a way that is efficient, has minimal impact on programs, and permits protected programs to cooperate by sharing some code and data items, while restricting access to others. The tension between these requirements was a key design consideration for many of the experimental and commercial computer systems created in the three decades since Dennis and Van Horn's influential paper. Two common and important goals are *uniform addressing*, meaning that data can be passed by reference among protected entities, and *modular protection*, meaning that access control choices are made independently for each data item. A number of addressing and protection models were developed to meet these goals, and incorporated into systems with protection features designed to be expressive, flexible, and easy to program.

Today, this diversity of approaches to protection is of mainly historical interest. Popular operating systems in 1995 either provide no protection at all, or provide protection through independent addressing environments in which the opportunities to share data are restricted. Today's operating systems are hosted by RISC processors using a relatively simple addressing and protection paradigm called *paged virtual memory*, incorporating the minimal hardware features needed to support them. While these systems have been quite successful, they lack the flexibility of the earlier models with uniform addressing and modular protection.

The shakeout of hardware and software models occurred for a variety of technical and market-related reasons involving portability, hardware cost, performance, and inertia. Today, general-purpose RISC processors are manufactured in large volumes: they are cheap and their designs are rapidly updated to incorporate technological advances. Either as cause or effect, few of the earlier systems emphasizing uniform sharing and protection survive today. A significant exception is IBM's AS/400 family [IBM88], perhaps the most successful computer system of all time. Even the AS/400 systems could advance in today's marketplace by migrating to "commodity" RISC hardware.

In this dissertation, I develop and explore a new model of addressing and protection that offers many of the advantages of earlier system models -- modular protection and uniform addressing of shared data -- while preserving the advantages of the models in wide use today: hardware and software that is simple, general, and efficient. I argue that compromises of previous systems were dictated by fundamental limitations of hardware technology at the time they were developed. Recent hardware advances -- in particular, support for large, sparse virtual memories -- present an opportunity to reconsider alternative models. The proposed model exploits these advances to support modular protection and uniform addressing within the framework of paged virtual memory. It can be implemented efficiently on standard RISC processors.

1.2 The Conventional Approach: Private Virtual Address Spaces

Paged virtual memory hardware permits an operating system to interpose a level of translation between *virtual addresses*, the data addresses generated by instructions executing in the processor, and *physical addresses*, which name locations in primary memory. The processor's *virtual address space* is the set of all virtual addresses for which the hardware is capable of representing translations. Privileged operating system software (the *supervisor*) controls the use of the processor's virtual address space by selecting the virtual-physical translations in effect at any given time. The supervisor uses this power to control both the interpretation of virtual addresses and the ability of user code to operate on data through those virtual addresses.

Most virtual memory operating systems use the processor's virtual address space independently for each executing program. The operating system provides a "meta-instruction" to create a new addressing context, or *process*, with a separate map of physical translations for the processor's virtual address space. The supervisor switches into a process by loading a reference to the address map into a protected register. In effect, these maps define a *private virtual address space* for each process. When a user program allocates memory or imports data, address regions are allocated from its address space and assigned to name the data within that process. Memory is protected by *address isolation*: all addresses are interpreted in the naming context of the issuing process, so it cannot illegally access the private data of any other process.

Private virtual address spaces increase the amount of address space available to all programs, since each process can use the entire virtual address space of the machine in its own way. However, private address spaces complicate sharing of code and data. Since virtual address bindings are assigned independently within each process, they have no meaning beyond the boundary or lifetime of a particular process. Virtual addresses are the most efficient and natural means of representing stored references, but they cannot be shared or transmitted between interacting processes without additional software support to translate addresses or coordinate address bindings from one address space to another.

The idea of multiple virtual address spaces evolved with early timesharing systems. Its purpose was to present a "virtual machine" abstraction to data processing jobs that were fully independent, self-contained, and competing for resources. In this context, there is little need for uniform addressing of shared data. Any shared data is simply copied between private virtual memories, for example, by reading and writing files in a well-defined format.

However, application structures have changed significantly since the early days of timesharing. Programmers today use high-level languages that insulate them from the details of the virtual machine; they do not care how the virtual address space is laid out. More importantly, in today's environments, individual programs are not fully independent. Modern software systems are often composed of cooperating components developed independently and packaged as separate programs, then combined to form a single integrated software system. The constituent programs may have rich interactions and protection relationships, for example, one process may control or pass inputs to another. Operating systems themselves often include user-mode programs that provide system services to applications, including operations on the data those applications consume.

This project originated from a study of *persistent stores* for computer-aided design and modern database environments [Cha90]. These systems implement shared repositories of pointer-based data structures by interpreting pointers and converting formats as data is copied between processes. In that study, I concluded that much of the cost and complexity of these solutions could be avoided with better operating system support. The difficulties are due partly to the private address spaces themselves, and partly to the view of computing that underlies the private address space model, in which virtual address spaces (processes) are units of software construction, execution, and resource management as well as protection.

1.3 Wide-Address Architectures

When this project began, it was clear that we were on the threshhold of a quantum leap in address space sizes. The appearance of IBM's RS/6000 [Mis90] and Hewlett Packard's PA-RISC [Lee89] had broken the 32-bit barrier using variants of segmented addressing. R4000 processors [MIP91] from MIPS Computer Systems and Alpha processors [Dig92a] from Digital Equipment Corporation entered the market in 1991 and 1992 respectively, completing the transition to flat 64-bit addressing and 64-bit datapaths. Several vendors are developing new wide-address processors: Sun Microsystem's UltraSPARC and Intel's P7 are rumored to provide 64-bit virtual addresses.

It is important to understand the magnitude of the shift that these architectures represent. The 21064 Alpha chip increases the amount of usable address space by three decimal orders of magnitude over Digital's VAX processors that were standard just a few years ago. Future implementations will increase this expansion to as much as ten orders of magnitude over the VAX, with no changes to the Alpha architecture. To illustrate the significance of this increase, consider that a full 64-bit address space will last for 500 years if allocated at the rate of one gigabyte per second and never recycled, while the 32-bit address space on the VAX and its contemporaries would last only a few seconds.

The premise of my research is that these wide-address architectures eliminate the need to overload virtual addresses in order to make sufficient address space available to applications, undermining a key motivation for private address spaces. To be sure, the immediate validity of this premise is open to question, given the terabyte-range address spaces available with the first 64-bit implementations. The critical point is that the appearance of these architectures crosses a natural threshhold, signalling a radical increase in the amount of address space available to operating systems and applications. This presents a rare opportunity to reexamine fundamental operating system structure, and to develop and explore new operating systems on the principle that virtual address space is no longer a scarce resource. Whether or not such a system is commercially viable today, it is clear that address sizes have leaped past the 32-bit boundary and will continue to grow. In the past this growth has averaged one additional address bit -- a doubling of address space -- every year [SBN82].

1.4 Opal: A Single Address Space Operating System

We set out to design a new operating system that exploits the power of wide-address processors to improve support for complex application systems. This system, called Opal, is based on a fundamentally different view of the relationship between addressing and protection. Crucial to the design is the separation of *address space* and *protection domain,* which are equated in the process concept of the private address space systems. An *address space* is the set of virtual address bindings for a particular software entity; a *protection domain* is the set of data items that an active code stream is permitted to reference through the virtual address space.[1] There is no useful distinction between these concepts in private address space systems.

Opal provides a *single virtual address space* shared by all procedures and all data. The fundamental principle of the Opal system is that virtual addresses are *context-independent*: they have a unique interpretation, for all applications, for potentially all time. The single address space can be extended to include data in long-term storage, and data across a physically distributed system such as a distributed memory multiprocessor, a workstation cluster, or even an enterprise network. This enables memory-mapped access to diverse storage resources in the network as a uniformly addressed single-level store. However, although executing user code can *name* all data in the system-wide virtual memory, its protection domain limits its access to a specific set of pages in particular modes (e.g., read-only) at any given instant.

The separation of protection from addressing is the cornerstone of Opal's emphasis on *orthogonal protection*, in which protection choices are independent of all other aspects of the system. In addition to decoupling addressing from protection, the Opal design also decouples program construction, program execution, resource ownership, and resource naming from protection domains. Opal has no conventional

1. We also use the terms *address space* and *protection domain* generically to denote the addressing context that defines the address space or protection domain, or code executing within that context.

"programs": all code exists as procedures residing in the shared address space. Protection domains are passive; procedures are activated using independent thread primitives. Common data resources too sensitive to share directly are accessed by protected procedure calls among protection domains; all such protected resources -- including instances of the Opal system abstractions -- are uniformly named and accessed, and their lifetimes are independent of the protection contexts from which they are used. Permission to access protected resources is passed through shared memory; all resources are shared by sharing memory. Any procedure can be an entry point for a protection domain. Subprograms can execute with or without a private domain, depending on the trust relationship between the caller and callee.

The intent of these choices is to eliminate any appearance of protection as a side effect of an unrelated decision (e.g., to execute particular piece of code), and any restrictive side effects when protection is used. Memory protection in private address space systems can appear as a side effect of a static choice to treat two software modules as separate "programs" -- a choice that could be made for a range of reasons involving code packaging, concurrency, modularity, or addressing limitations. Once this choice is made, it restricts the means by which those modules can interact.

Opal's goals and concepts are related to those of many previous hardware and software systems spanning over 25 years, including Multics [DD68], Hydra [Wul74], Cedar [SZBH86], Pilot [RDH+80], Monads [RA85], Intel iAPX/432 [KCD+81], IBM System/38 [HSH81b] and AS/400, Bubba [CFW90], Psyche [SLM90], and several other systems. Opal differs from these systems in various respects, for example, in the granularity of protection, the manner in which protection is provided, the relationship of protection to addressing, and the division of function within the system. These systems and their relationship to Opal are discussed in Chapter 2 and Chapter 3.

1.5 Structure and Contributions of the Dissertation

The single address space model presented in this dissertation is conceptually important because it changes a basic operating system axiom -- that protection domains and address spaces are identical -- and explores its implications. As a practical matter, it presents opportunities and tradeoffs at all levels of the system, from memory system hardware architecture up through the structure of the system and its applications. This dissertation describes experimental research to explore the viability and implications of the single address space approach, encompassing the following stages and contributions:

- Design of an operating system with a single virtual address space and multiple protection domains, consisting of new kernel primitives for protection, communication, virtual storage management, and resource control. Chapter 3 and Chapter 4 present the Opal system model and interface, set it in context with other systems, and outline its strengths and weaknesses.
- Design of an object-based sharing and execution model for Opal, comprising runtime system support for *shared objects* and *protected objects* above the foundation of a global virtual memory. This package, described in Chapter 5, is the basis for uniform access to shared data and resources in the Opal system.
- A prototype implementation of the Opal kernel primitives and runtime environment, outlined in Chapter 6. The prototype consists of a subsystem server and runtime package running above a standard microkernel (the Mach system from CMU) on 64-bit Digital Alpha AXP workstations. Microbenchmarks demonstrate that Opal's low-level primitives are implemented with performance commensurate with other operating system environments built on the same hardware systems and microkernels.
- Demonstrated use of the Opal primitives to transparently reconfigure memory protection for cooperating software components. Section 6.4 introduces an Opal-based implementation of software *mediators* [SN92], a structuring paradigm for integrated software systems. Protected objects and mediators permit alternative protection structures, including shared memory, with minimal impact on modular application code. Simple experiments with the mediator facility show that applications can be reorganized under Opal to improve their performance relative to common organizations based on copying shared data between private virtual memories.
- Design and prototype of code handling and linking facilities for the Opal system, based on a new approach called *global static linking*, which permits dynamic sharing of statically linked procedures, without the expense of dynamic linking. Chapter 7 presents these facilities, and illustrates some of the tradeoffs of the single address space model by examining the handling shared and private symbol references in shared procedures.
- Design and partial prototype of extensions to Opal for persistent and distributed data, described in Chapter 8. These form the core of a new foundation for workgroup cluster computing in which networked workstations share a global virtual memory, as a substrate for cooperation and as a repository for long-lived data.
- Qualitative analysis of the factors that determine the practicality of the single address space model, including fundamental limitations of the model and their impact. Section 3.3 provides an overview of the key issues.

Despite the experimental nature of this work, quantitative metrics are of limited value for evaluating the result. Opal is the first operating system to support a protected single virtual address space on standard 64-bit RISC processors. Its primary purpose is to show that the longstanding goals of uniform addressing and modular protection, which previously required unacceptable tradeoffs in terms of hardware and software complexity and generality, can now be achieved in an efficient and general way by exploiting recent hardware advances. The single address space structure enables new uses of protection and sharing that are useful and transparent to modular applications. However, an evaluation of the degree to which this new flexibility meets real application needs must be based on experience with large-scale application systems in actual use. This would include a quantitative comparison of the approach with other means of attaining similar goals, for example, software-based protection and addressing schemes. Such an analysis would be premature, and this dissertation does not provide it.

The primary contributions of this dissertation are: (1) a complete definition of a functioning system that addresses the issues raised by the single address space concept, (2) a prototype to demonstrate feasibility of the model and serve as a basis for future experiments, (3) an enhanced understanding of the limitations of both private and single address space structures, and (4) an application framework that exploits the flexibility of the single address space model, and can be adapted to support some of its features in conventional systems.

I now summarize the basic argument of this dissertation:

- The *private address space* operating system models popular today are based on outdated notions about the relationship between protection and addressing, which were formed and accepted at a time when self-contained applications and small virtual address spaces were the norm.
- VLSI advances have made it possible to support very large virtual address spaces cheaply, producing commodity RISC processors powerful enough to efficiently support a *single virtual address space operating system* with a protected global virtual memory. This is demonstrated by a prototype.
- The single address space model attains the key goals of *uniform addressing* and *modular protection* that were common to many earlier systems, while preserving the simplicity, generality and performance responsible for the commercial success of the current models.
- Uniform addressing and modular protection are the “right” goals: for complex software systems, the single address space model enables structuring choices that are superior to choices widely made today in terms of performance, reliability, or extensibility.

Chapter 2

Memory Protection and Software Structure

This chapter examines operating system models of addressing and protection, and their facilities to allow protected application entitities to interact. The purpose is to motivate the single address model and set it in context. Our discussion is grounded in a view of computing in which complex applications are constructed from components that are produced independently and tied together through operating system primitives for sharing, protection, and communication. These *integrated systems* present challenges for operating systems because the components are autonomous and can benefit from protection, yet they must work together and share information efficiently.

Section 2.1 presents and justifies the integrated systems model. Section 2.2 then discusses the interaction paradigms of four classes of operating systems: (1) *mono-address space* systems with no hardware-based memory protection, (2) *private address space* systems using stream communication, (3) *object-based* systems using messages or protected procedure call, and (4) *segmented* systems using shared memory. We argue that protected memory sharing in the segmented systems can aid the construction of efficient integrated systems, and that the single address space model simplifies this style of interaction by providing stable and uniform addressing of shared data. Finally, Section 2.3 explores alternative schemes for addressing shared data, including several architectures that support global addressing in hardware.

2.1 Integrated Systems

Any large and evolving software system can be profitably structured as a collection of components that interact in well-defined ways. Components can be added, removed, or modified easily, so long as the defined interactions are minimally affected. An *integrated system* is a collection of component programs that work together, for example, to directly support users in some complex task, such as developing software or writing a thesis. From a systems perspective, "integrated" means that the components have rich, fine-grained interactions that must be supported efficiently.

An example of an integrated software system is a CAD/CAE environment used by the Boeing Company in aircraft design. This system was presented to us by members of Boeing's High Performance Design Systems group, a research and development group experimenting with database and operating system technologies for next-generation CAD systems. Boeing's CAD system consists of collections of programs ("tools") that operate on stored data structures representing aircraft designs and derived information. These tools interact in part by reading and writing a shared CAD database. Distributed access to complex, persistent data structures is fundamental to this application, but we defer our discussion of those issues until Chapter 8. In this chapter we are concerned only with protection and interaction within a single node.

Boeing's CAD system is continually evolving as Boeing engineers develop new ways to process the "hundred-gigabyte gold mine" of aircraft CAD data to refine their designs and reduce the costs of manufacturing, maintaining, and operating their aircraft. These extensions are added to the system as new CAD tools. The current system includes tools for visualizing the aircraft design, verifying design constraints, simulating behavior, standardizing parts, planning manufacturing steps, and routing hydraulic and electrical connections. These tools import information about the aircraft design in standard formats. Many of them derive new representations of design data, which are then consumed by other tools.

We view these systems as consisting of *components*, which are discrete, identifiable units of software construction and instantiation. The exact nature and granularity of a "component" is left unspecified; the purpose of the term in this chapter is to give us a vocabulary to explore protection structuring choices. Statically, components are modules -- collections of procedures implementing specific functions on data structures in a specific format -- but at runtime we might think of multiple instances of a module, operating on separate data structures, as separate components. Tools are built from components, and might themselves be viewed as components. Two tools might interact by invoking a common component. Components might have procedures and data in common. For example, several components might perform different functions on structures of the same format, including transforming that data to other formats processed by other components. The partitioning of a complex system into components is a design choice of its creators, but the system must provide a means to protect components and program their interactions efficiently.

System support for interacting components is also important in the document-centered world of personal computing. Traditionally, documents were managed by monolithic applications supporting a full range of functions, permitting the user to create, edit, display, and print the documents managed by that application. As the software advances to graphical user interfaces, it becomes natural to cut and paste from one document window to another, creating a single document with, say, embedded figures created by a drawing program, spreadsheets created by a table editor, graphs generated from spreadsheets, as well as sound, text, and images. The user wishes to store, manipulate, print, and display the diverse items in the document as a unit, yet the procedures that perform these functions on each item reside within the application that created the item. Thus a complex document combines procedures and data formats from multiple applications, which must interact to manage the document.

Components may be independently authored and perhaps sold by different vendors. Users should be able to combine them in ways that were not foreseen by their developers. An *integration framework* standardizes the interaction among components so that new components and data formats can be "plugged in" to an existing body of software. For example, Microsoft's Object Linking and Embedding [Cla92] allows users to combine separately purchased components to support complex documents. Components have little or no explicit code to deal with each other, and software vendors can ensure that their software is compatible with other components written with standard data formats and rules for interaction. Each vendor focuses on building and selling components that implement specific functions well, leaving users to buy other functions from another vendor. This creates a competitive market for components, and enables the end user to purchase the specific set of components best suited to his or her needs.

In summary, integrated systems have unique needs that must be considered when evaluating operating system facilities for protection and interaction. Many common uses of computers today involve the design of artifacts such as complex documents, computer software and hardware, media streams, and aircraft designs. These artifacts are represented by collections of long-lived, complex data structures in defined formats, which are processed in various ways by collections of interacting software components. Advances in software technology have made it possible to construct complex application systems by combining separately developed components, and change them incrementally by modifying specific components. These application systems rely on operating system support for component interactions that are both safe and efficient, and that can be extended to accommodate new component interfaces and data formats.

2.2 Operating System Models of Addressing and Protection

The view of software systems as collections of cooperating components is becoming more prevalent as new application areas are developed and as older tools are integrated by gluing them together through integration frameworks and shared data repositories. Central to this view is the idea that components are separately authored programs, possibly running on behalf of different users. This raises the question of what services the operating system should provide to allow components and their data to be protected from one another, while permitting the rich data sharing needed for efficient component interactions. We now consider several operating system models in terms of their support for this application style. These systems form a continuum ranging from “efficient, flexible, and unprotected” to “expensive, restrictive, and protected”. Our purpose is to expand the choices in the middle range of this continuum.

2.2.1 Mono-Address Space Systems and Software-Based Protection

The majority of computer systems in use in 1995 (e.g., those based on Microsoft’s DOS [Mic91]) provide no means of isolating coresident applications; any active software entity can address the entire physical memory of the machine. In effect, all software executes within a single address space and a single protection domain. DOS and other unprotected systems were designed for single-user personal computers that first became available in the mid-1970s, before virtual memory facilities were available on low-end platforms. They are flexible in that separately developed application components interact freely, simply by calling each other’s procedures. This structure is depicted in Figure 2-1 to introduce some notation used to represent interacting components graphically. Rich application systems have grown around them. However, malicious or accidental malfunctions can easily propagate from one software component to another. As a result, unpredictable system failures, viruses, and the threat of data corruption are a fact of life for most computer users.

Language-level access checking can provide some degree of fault isolation in a system with no hardware-based memory protection. Some mono-address space systems rely on the type checking and scoping rules of a “bulletproof” programming language, as a cheaper and more efficient alternative to memory protection at the hardware level. For example, the Xerox Pilot [RDH+80] and Cedar [SZBH86] systems support a single virtual address space for applications written in a system-mandated type-safe programming language. Emerald [JLHB88], Guide-1 [BDA+91], and Hermes [SBG+89] are distributed programming languages designed to support cooperating applications within a shared addressing context on each node. More recently, there has been a rebirth of interest in language-based protection, both as a means of supporting safe runtime extensions to a protected programs (such as operating system kernels [BSP+95] and database servers[Mye93]), and as a means of safely importing untrusted code through the Internet for execution by a trusted interpreter.

Other software protection schemes use a trusted compiler or postprocessor to inject *sandboxing* instructions into executable code, to limit its access by bounds-checking memory references [WLAG93], with a moderate execution-time penalty. It has also been argued that components randomly placed at “anony-

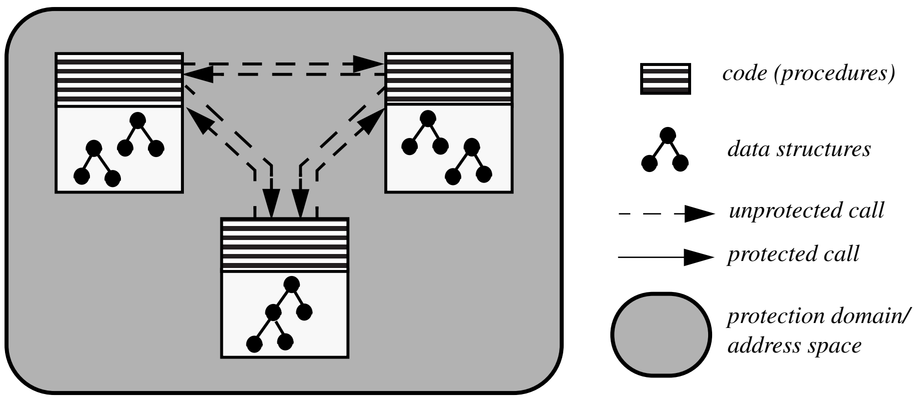


Figure 2-1: Unprotected components in a mono-address space system.

mous” locations in a full 64-bit address space are protected because no component can “guess” where another’s data is located [YBA93].

These and other proposals for software-based protection are intriguing in part because protected components can cooperate without hardware protection context switches, which are expensive on current architectures. These approaches may coexist alongside hardware-based memory protection as potentially lower-cost alternatives, but they do not eliminate the need for the stronger and more general isolation provided by hardware-based memory protection. For example, while it is true that safe languages permit a relaxation of hardware protection, *exclusive* reliance on language-based protection precludes the use of popular and efficient programming languages.

2.2.2 Private Address Space Systems

The problems with unprotected mono-address space systems dictate that personal computer operating systems evolve to support hardware-based memory protection. This is not in dispute: the next generation of desktop operating systems (e.g., Microsoft’s NT [Cus93]) incorporate virtual memory protection in some form. Virtual memory management features are now supported on all modern general-purpose processors, and they are the only reliable means of enforcing isolation across software written in multiple languages.

In their protection models, these new systems follow the *private address space* operating systems, such as Unix variants and VMS [LL82]. Protection is provided by running each program as a separate *process*, comprising a private virtual memory with one or more code streams (*threads*) executing within it. The private address space structure evolved from the early days of timesharing, when the purpose of the operating system was seen as sharing expensive physical resources fairly and transparently among self-contained batch jobs. Shared files are the primary communication mechanism.

While processes are the primary abstraction for an executing program, private address space systems provide ways to combine multiple components in a single process. Components can be statically linked into the same program image, or packaged as dynamic code libraries instantiated as needed within each process. With these protection structures, components can cooperate easily by invoking each other's procedures and passing data by reference. However, protection, safety, and reliability are compromised. A component that behaves erratically can damage the private data of another component; although it has no legitimate need to reference that data, the protection structure does not prevent it from doing so. Software failures can easily propagate unless some form of software-based protection is used.

Components can be protected by instantiating them in separate processes that interact with system communication primitives, such as *byte streams* (depicted in Figure 2-2) or *messages* (discussed in the next section). For example, Unix has well-defined rules for programs to interact by generalizing the file abstraction. The producer of shared data *linearizes* [HL82] internal data structures into an unstructured stream of bytes, which is copied into a system buffer. The stream may be stored in a file, but is eventually read from the system buffer by a consumer, which "unpacks" the bytes into some chosen internal data structure format in its private virtual memory, discarding the bytes read from the stream. The producer and consumer may in fact be the same program executing at different times.

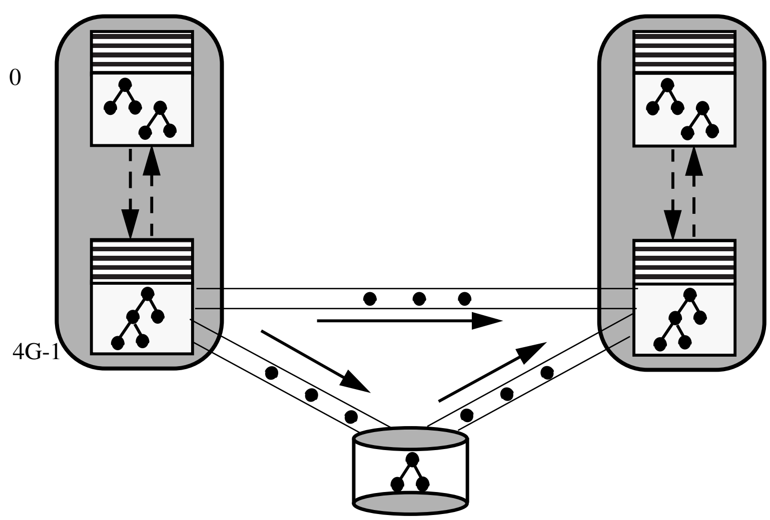


Figure 2-2: Components in private address spaces interacting through a stream.

The advantages of streams are: (1) for some components, the intermediate representations are easy to standardize as machine-independent or even human-readable formats, (2) each component can choose an optimized internal representation, and (3) data structures can be validated during conversion, so the consumer fails cleanly if its input is malformed. However, stream integration can involve a significant amount of unnecessary or redundant effort:

- Useful information maintained by the producer is discarded and recomputed by the consumer if it cannot be represented in the predefined stream format.
- Useful information maintained by the consumer is discarded and recomputed each time a new version of the input stream is read, even if only a part of the dataset has changed. A consumer that needs only a part of the information in an unstructured stream generally must scan the stream sequentially to get it.
- The performance of packing, unpacking, and transmitting streams is limited by memory system bandwidth and is becoming more expensive relative to integer performance as processor cycle times continue to improve faster than DRAM latencies [ALBL91]. Large streams exchanged through scratch files on disk generate unnecessary I/O.
- Stream integration can lead to a proliferation of *ad hoc* data formats used by each application, resulting in additional overhead to convert between formats.

The fundamental problem with streams is that they do not support incremental access to complex data, limiting the ability of integrated systems to scale to larger datasets. To support incremental access, internal data structures can be represented in a seekable file using offsets or software-interpreted pointers. However, such structures are cached redundantly in both the producer's and consumer's virtual memories, introducing overheads to maintain the copies and keep them consistent. Reading and caching streams imposes virtual memory overheads to zero-fill pages for the consumer's copy and maintain them on backing store; a virtual memory system will treat them as "dirty" even if they have never been modified by the consumer.

The legacy of Unix is that streams are the dominant medium of data exchange between protected application components today. It is natural for private address space systems to share data by copying between private virtual memories, and this is a simple and clean solution if processes are data processing jobs that read some input, compute a result, and terminate. However, streams do not support incremental updates and incremental access to complex data structures, a more natural and efficient model for cooperating components in information-rich application areas, e.g., aircraft CAD.

2.2.3 Coarse-Grained Objects with Messages or RPC

An alternative to streams is to model all interactions through protected messages or protected procedure calls. Interactions are "protected" when the sender and receiver (caller and callee) cannot access internal data structures of the other, and each side chooses the code it executes. Messages or arguments and results are passed in agreed formats, and are validated by the other side before use. Messages and cross-domain procedure calls are appealing in part because they extend naturally to distributed systems. For example, Remote Procedure Calls (RPC) [BN84] are widely used to construct distributed services, and are so well-known that calls across memory protection boundaries are often referred to generically as "RPC calls" even when no network communication is involved.

The model of protected software entities interacting through messages is expressed most generally in *object-based* systems. The idea of "objects" as a structuring paradigm is a recurring theme of this dissertation and is central to many systems related to Opal. Generically, an *object* is a data construct whose representation is hidden behind a procedural interface; each object is manipulated only by invoking a specific set of procedures (*methods*) defined by the object's *type*. Every object is named by a unique reference; objects may be passed by reference to the methods of other objects, and references may be stored within objects to form linked structures of arbitrary complexity. Objects capture the notion of "plugging in" components because objects are uniformly named and their methods are dynamically bound: given an object reference and an interface, a program can operate on the object by invoking methods that "come with" the object.

We will have much to say about different styles of objects later (Section 3.2), but at present we are concerned only with systems that model all protected entities as objects. Object-based operating systems, such as Hydra [Wul74], Eden [ABLN85], Argus [LCJS87], and Clouds [AM83], center on *protected objects* whose encapsulation is enforced by hard memory protection boundaries. A protected object reference is an unforgeable *capability* [Fab74] that confers limited privilege to operate on the named object in specific ways, by protected procedure call to some subset of the object's methods. Protected objects in these systems are *coarse-grained*: they are implemented as separate virtual address spaces, and are equivalent to protection domains. While the uniform capability space can be thought of as a higher-level "address space" of objects, each object can interact only with the objects named by capabilities that are accessible through its virtual address space, and only by making RPC calls to their methods.

In coarse-grained object-based systems, components or groups of components can be protected by modeling them as objects interacting by messages or RPC. In contrast to streams, this model allows modular incremental access and updates to shared data. For example, any change to the state of an object can be propagated by invoking methods on other objects, which can respond to the notification by incrementally updating their own internal state. However, cross-domain messages and RPC are at best an order of magnitude more expensive than ordinary procedure calls, despite recent optimizations [e.g., BALL90, Lie93].

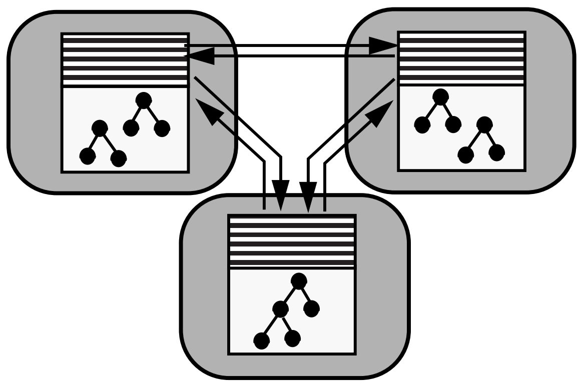


Figure 2-3: Components interacting as coarse-grained protected objects.

2.2.4 Shared Segments

To this point we have visited two basic classes of systems: those with no hardware-based memory protection, and systems with protection based on objects or processes encapsulating fully-isolated software components and their data. The unprotected systems permit interaction through shared memory, which is efficient, implicit, asynchronous, and incremental. While the protected systems are more reliable, interactions across protection boundaries are more expensive in these systems, because they rely on data copying and protection context switches.

A solution to this dilemma is to relax the model of fully disjoint protection by allowing protection domains to overlap, sharing direct access to data in virtual memory. Dynamic sharing is a key aspect of Multics and other systems using segmented addressing, for example, the IBM 801 [CM88] and RS/6000 [Mis90] architectures. These systems support efficient sharing within a single virtual memory partitioned into *segments*, each with a separate address map. A user process gains access to a segment by issuing a protected *load segment* system call; the supervisor checks authority and loads the segment's map into one of a vector of protected *segment descriptors* (or registers) that are part of the process addressing context. Segment data is addressed as an offset from an indexed segment descriptor. Many segmented systems support access to data in long-term storage through virtual memory segments; the system handles memory references to a *mapped file* segment by reading and updating pages of the file. This feature is often called a *single-level store*; some of its well-known advantages are discussed in Chapter 8.

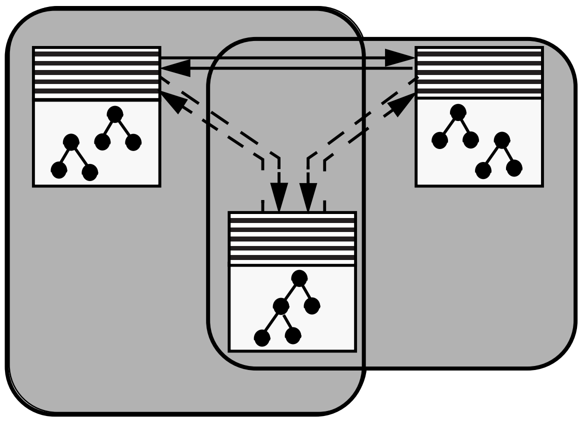


Figure 2-4: Protected components interacting through shared code and data.

Operating system support for shared memory and mapped long-term storage is central to this dissertation. These features permit application components active at different times, in different locations, or in separate protection domains to interact by sharing and exchanging linked data structures, accessed incrementally in virtual memory. While direct memory sharing is generally not as "safe" as no sharing, judicious sharing of virtual memory often presents an acceptable tradeoff of some protection for a significant improvement in performance.

- Asymmetric trust relationships are common and can be exploited: ***A*** might accept inputs (or memory segments) from ***B*** even when ***B*** does not trust ***A***. For example, a name server can provide read-only access to its database, requiring protected calls only for updates; we revisit this idea in Section 5.3.
- Memory shared among mutually suspicious components can be mapped read-only, used in restricted ways, or passed sequentially (for example, by *page remapping* [DP93]) and validated before use.
- Protection domains can be used to coordinate data access among mutually trusting threads within a single component. For example, multiple instances of a database program may execute in separate domains to enforce different access privileges, or to use protection faults to drive data management using one of the techniques described in [AL91]. Protection-driven locking for atomic transactions on shared segments is a key feature of the IBM 801 and RS/6000 systems.

- Tradeoffs between protection and performance are unavoidable. Complete isolation can never be achieved: even if protection domains are fully disjoint, fundamental granularity tradeoffs must be made. Also, programs may have naturally overlapping access to stored data, and this may not be known in advance. Overlapping protection domains can be used to enforce the principle of least privilege (“need to know”) when sharing is permitted.

For the integrated systems discussed earlier in this chapter, memory sharing among protected entities offers an opportunity for application components to efficiently and incrementally propagate changes through a collection of data structures representing design artifacts and derived information. Components that process common data formats can directly access a shared copy of their input in memory. Components updating derived data stuctures can access the source structure directly in memory. Components operating on a fragment of a larger structure can be forced to limit their access to just that fragment.

2.2.5 Addressing Shared Data

One problem with sharing segments using segmented addressing is that virtual address pointers do not have a uniform meaning across the system. While a segment has a global virtual address, programs do not use these global addresses directly. Each process sharing a segment maps the segment into a private address space, perhaps at a different virtual location (e.g., using a different segment descriptor). As with private address space systems, this can complicate sharing or exchange of data containing addresses. Procedures must include indirect addressing structures to manage the multiple address assignments of shared data segments on which they operate. For example, in the CPR/801 system, the PL/1 language defines a special *refp* type for a pointer to a shared segment, with conversions handled by the language runtime system. Programs can represent internal pointers as segment offsets, but the compiler must make a static choice to handle these pointers differently. Cross-segment pointers are fragile.

Shared memory and mapped files, which are central to segmented systems, have appeared in many private address space systems (e.g., [GMS87]) as an exception to the rule of fully disjoint protection. However, there are pitfalls and limitations to their use within the private address space model. Use of pointers typically requires explicit *a priori* coordination of address space for shared regions to ensure that shared regions are addressed uniformly. A program that is the sole user of a mapped file can easily reserve a fixed address range for file mapping, but this does not extend to multiple processes or regions; in general, sharing patterns must be known statically.

Recent systems have taken steps to resolve this problem by arranging for the *system* rather than the applications to coordinate address bindings, in order to handle dynamic sharing patterns in a uniform way.

Examples include *based sections* in Microsoft NT [Cus93], the "shared" options to file mapping primitives on recent Unix systems (e.g., *mmap* with MAP_FIXED on OSF/1), and the *segment loader* in Lightweight-RVM [SMK+94]. Even in these systems, however, the mix of shared and private regions can be difficult to manage. Pointers may be ambiguous: private data pointers in shared regions are difficult to detect and handle, and private code pointers (important for object-oriented languages) cannot be passed or shared. Sharing is *ad hoc*, since any process that will *ever* use a shared region must have the assigned virtual addresses free in its address space.

These efforts are steps in the direction of a single virtual address space. In the next chapter, we argue that fully global coordination of address space is an attractive operating system choice on unsegmented RISC processors. This approach eliminates the problems with nonuniform addressing by treating *all* virtual address space as a global resource controlled by the operating system, like disk space or physical memory. This is the essence of the single address space model. Before introducing the details of the model, we first consider architectures that provide specific hardware support for a protected common address space.

2.3 Global Address Space Architectures

Our emphasis on uniform addressing as a basis for supporting sharing and cooperation is common to many other systems. In this section we outline the distinctive characteristics of several representative systems with specific hardware support for uniform addressing and protection, including *capability-based* hardware systems surveyed in [Lev84]. The intent is to sample the broad range of possible hardware features to support globally uniform addressing, and to give a sense of what is gained and lost by building a single address space system with a paged virtual memory in the absence specific hardware support.

2.3.1 Global Segment Addressing: IBM System/38 and AS/400

IBM's AS/400 system [IBM88] and its predecessor the System/38 [HSH81a] are capability-based architectures with global addressing and segmented protection. User data is allocated from *space objects* equivalent in granularity and other key respects to segments on the RS/6000. Pointers can be passed or shared on the System/38 and AS/400; any entity that obtains a pointer automatically obtains permission to access the named segment in specific modes permitted by the pointer (a subset of read, write, execute, update). Addresses are never recycled, so dangling references cannot occur. Pointers are tagged and their integrity is enforced by the machine; pointer arithmetic is permitted, but only within the boundaries of a segment.

2.3.2 Object-Oriented Addressing: Intel iAPX/432

The Intel iAPX/432 [Org83] was the product of an ambitious effort to provide hardware support for the Hydra operating system model, with full support for object-oriented capability-based addressing. The 432 system also included support for a uniformly-addressed object filing system [PKW81].

The 432 protection hardware controlled access to memory at an object grain. The protection model was similar to the coarse-grained object systems discussed in Section 2.2.3, except that the granularity of protection was at the level of programming language records, rather than application components or operating system services. The Ada compiler modeled every Ada object as a protection domain, and every procedure call as a protected procedure call. This resulted in a system that was expensive to design, build, test, debug, purchase, and use.

2.3.3 The Monads-PC Architecture

The Intel 432 and other capability systems suffered from (among other things) an insufficient underlying hardware base. For example, addressing on the 432 was not fully uniform, due in part to the way the processor address space was managed: capabilities contained only 24 address bits, which were translated to and from 80-bit UIDs as objects moved between memory and persistent storage. Because the underlying virtual address space on the physical hardware was too small, the result has always been an *emulation*, at some level, of a large address space system on a small address space machine.

The Monads architecture [RA85,Ros92] resolves this problem with by supporting capability-based addressing and protection above a flat 60-bit paged virtual address space that includes persistent data and spans a local network. Above its paged address space, Monads provides an object-based addressing and protection model supported by special-purpose hardware, including capability registers. In this system, the purpose of the single address space is to streamline the implementation of capabilities and to extend the capability space across a network.

2.3.4 Guarded Pointers and Other Proposals

Dally et. al. [CKD93] propose protection using *guarded pointers*, a restricted form of capability, in part to reduce the cost of context switching on latency-tolerant multiprocessors, which context switch frequently to hide memory latency. We believe that guarded pointers would offer useful hardware support for Opal, eliminating some compromises of our present Opal implementation. Okamoto et. al. [OSS+92] propose a single address space architecture in which memory access privileges are determined from the address of the instruction that issued the reference. This approach may offer cheap hardware protection for software that is statically partitioned into fully disjoint protected modules. Silberschatz and Ozden [SO92] propose hardware support for segmented protection in a single virtual address space architecture.

2.3.5 Hewlett-Packard PA-RISC

The Hewlett-Packard PA-RISC [Lee89,Hew90] is a leading RISC architecture that is a hybrid of segmented and global addressing. Like the segmented hardware systems, the PA-RISC has hardware registers (*space registers*) to extend the addressability of applications using 32-bit pointers. However, user code can load any value into a space register with an unprivileged instruction, enabling full global addressing, with sophisticated protection features (Section 3.1.4). Thus the PA-RISC is effectively a single address space architecture, if long-form addresses are used.

Most software on the PA-RISC uses short-form addresses, because they are more compact and efficient, and because they permit backward compatibility with private address space operating systems. However, the PA-RISC is well-equipped to support a single address space operating system such as Opal.

2.4 Summary

We have evaluated four classes of systems with respect to how their models of addressing and protection support integrated software systems with interacting components.

- *Mono-address space* systems with no memory protection. In these systems, cooperation is simple and efficient, but they are limited in their ability to scale to larger function sets, because failures can easily propagate between components. These systems can support various forms of software-based protection, for example, use of a single type-safe programming language.
- *Private virtual address space* systems composed of processes or objects communicating by streams, messages, or RPC. These systems with fully *disjoint protection* cannot easily scale to larger data sets due to the costs of integration based on copying and protection switches between their private virtual memories.
- Systems with *overlapping protection*, including segmented systems and private address space systems with shared memory and mapped files. Direct sharing supports incremental access to large, rich information structures. However, nonuniform addressing complicates the use of shared segments in these systems.

- *Global address space architectures* that provide hardware support for uniformly addressed shared data, and modular protection within a global address space. We have included in this category the various flavors of capability-based systems, as well as several popular processors that support interesting protection features within a virtual address space.

These systems present a wide range of choices regarding the relationship between addressing and protection, granularities of sharing and protection, uniformity of naming, and the level of the system at which important features are implemented.

We summarize the situation somewhat starkly as follows. We contend that operating system models in common use -- providing either no protection or fully disjoint protection -- encourage the development of software systems that are slow, unreliable, or poorly integrated. An application designer has two basic choices: represent application components as independent address spaces that exchange data through pipes, files, or messages, thereby sacrificing performance, or place multiple components in one address space, sacrificing protection. Many systems have taken steps to resolve this dilemma by permitting sharing or transfer of virtual memory among protection domains. Of these systems, some are based on addressing models that do not fully support general, uniform sharing, while others support uniform sharing with specific hardware features that compromise their performance and generality in other respects.

Also in the picture are leading processor families, such as the PA-RISC and other RISC processors, whose features are flexible enough to support several addressing and protection models, including variants of the single address space model we propose. Finally, we noted several proposals for incremental extensions to current architectures, designed to support efficient protection within a single address space. These include guarded pointers and the Monads-PC.

The point of this dissertation is that we can support uniform sharing and protection *without* special hardware features on 64-bit RISC architectures, given an appropriate operating system structure. In doing so, we develop an operating system model that can exploit architectures that do provide explicit support for protected global addressing. We now turn to the single address space model.

Chapter 3

The Single Address Space Model

This chapter presents the single address space model and its relationship to the systems described in the previous chapter. We propose Opal's single address space model as an operating system choice on standard processors designed to support multiple address spaces (e.g., Unix). Like conventional private address space systems, Opal provides a paged virtual memory model with multiple protection contexts (protection domains). However, all virtual addresses are context-independent: Opal's addressing contexts control access to a global virtual address space, rather than establishing a private naming environment, as in private address space systems. The purpose is to support modular protection and sharing for interacting software components, and to naturally accommodate persistent and distributed sharing (Chapter 8). In Opal, data can always be passed by reference when protection choices allow.

Opal combines several key ideas from the systems discussed in the previous chapter. The elements of the Opal model are (1) a uniformly addressed global virtual memory, (2) "pure" protection domains, (3) segment-grained allocation, sharing, and management of virtual memory, and (4) uniform naming and access control for shared segments using software capabilities. Our goal is to demonstrate that recent advances in hardware, operating systems, and language technology enable us to achieve the goals of many earlier systems, but without the need for special-purpose hardware, without loss of protection or performance, and without relying on a single type-safe language.

This chapter gives a "bottom up" overview of Opal's single address space model and some of the issues involved. Since good performance on commodity hardware is a primary goal, Section 3.1 begins by exploring the question of how a large, sparse, protected virtual memory can be handled efficiently on today's RISC architectures. Section 3.2 then outlines Opal's approach to managing a global virtual memory in a general and efficient way. Section 3.3 gives an overview of the tradeoffs involved in the single address space structure.

3.1 A Protected Global Address Space for Paged MMUs

Today's operating systems implement their addressing and protection schemes using virtual memory facilities provided by the paged memory management units (MMUs) on all modern general-purpose processors. The MMU hardware accepts virtual-physical translations loaded by the supervisor, grouped in fixed-size *pages* (e.g., 8K bytes) to amortize the overhead of storing the translations. The supervisor may enable and disable specific translations whenever it gains control, e.g., during context switches and system calls. If user code attempts to reference a virtual address for which there is no active translation, the MMU hardware raises an exception to return control to the supervisor, which determines how to respond to the event. For example, the supervisor may install the required translation and restart the faulted instruction, or it may report the event to user code.

Most aspects of virtual memory management are unchanged by the single address space structure. The fundamental difference is that the supervisor maintains a single global set of virtual-physical page translations, rather than separate translations for each addressing context. The key challenges are (1) efficiently managing the large, sparse set of translations implied by a global address space, and (2) supporting memory protection within the global address space. The global address space architectures discussed in Section 2.3 supported protected global addressing in some direct way, e.g., using tagged or guarded pointers. Our goal is to support a similar sharing and protection model on standard RISC architectures. In this case, system software must support a sparse global address space and page-grained protection using MMUs designed primarily for systems with multiple virtual address spaces, which protect memory by address isolation. In a single address space system on stock hardware, protection must be enforced not by enabling separate sets of context-specific translations on each context switch, but by *disabling* global translations for pages that are not in the protection domain of the executing instruction stream.

Recent RISC architectures have shifted more memory management functions into software, granting the operating system supervisor more control over memory management. This control can be exploited to support a protected global mapping efficiently, including high-performance context switches between protection domains. MMUs differ in the details of how translations are enabled and disabled. We discuss these differences in some detail to clarify Opal's addressing and protection model, and the practicality of its implementation on standard processors.

3.1.1 The Trouble with Hierarchical Page Tables

We begin by discussing traditional MMUs and some of the problems they raise. Some processors (e.g., VAX[Dig81], SPARC[Cyp90], x86 [Int88]) have microcoded MMUs that fetch missing translations directly from address maps maintained by the supervisor in memory. On these processors, the operating system supervisor must represent address maps in the architecturally defined formats, typically variants of the linear or forward-mapped hierarchical page tables used on the VAX and other early virtual memory architectures. These mapping formats efficiently represent multiple compact address spaces, using a tree-structured table of virtual-physical mappings for each addressing context.

Hierarchical page tables are not well-suited to the single address space model. In these systems, an address is accessible if and only if a valid translation exists in the address map loaded on the processor. A single address space system has a single global set of translations; it should follow that such a system would use a single shared address map. However, it is prohibitively expensive to implement protection by enabling and disabling translations in the global address map on each protection context switch. Instead, a single address space system on these architectures must maintain separate address maps for each protection domain, essentially representing the single virtual address space as a collection of private address spaces, each defining one protection domain's "window" on the global virtual memory.

With this arrangement, a protection context switch is equivalent to (but no faster than) a process switch in a private address space system on the same hardware. However, there are several flaws. First, this scheme redundantly stores translations for shared pages in each map. (But note that in some systems, a large, aligned shared region can be mapped through a shared leaf map.) More importantly, the space overhead of hierarchical page tables can be very large if the active pages are scattered widely in virtual memory, which is likely if all data resides in a single address space. The maps can be arranged so that large, aligned holes consume no space in the maps, but the space overhead is prohibitive in the general case. In short, a single address space is a poor idea for architectures requiring hierarchical page tables.

3.1.2 Software-Loaded TLBs and Hashed Page Tables

Thus conventional hierarchical page tables are poor for large, sparse virtual address spaces, and for sharing. They seem unlikely to survive, even for private address space systems. However, alternative structures are available and can be used on many RISC processors, including the MIPS and Alpha architectures, that export more control over address mapping to the supervisor software through the use of *software-loaded TLBs* for memory management. The translation lookaside buffer (TLB) is a cache on the processor chip that holds recently used translations. The processor completes a virtual-physical translation in a single cycle if the requested mapping is resident in the TLB. However, if the mapping is not resident in the TLB, an MMU with a software-loaded TLB traps to supervisor software to retrieve the missing translation. The software may use any data structure to retrieve the mapping and load it into the TLB; processors with software-loaded TLBs do not impose architecturally defined translation table formats.

For Opal, we must select a data structure that handles a large, sparse, global virtual memory compactly and efficiently. We propose a global *hashed resident page table* (HRPT) similar to the structure used to fill the TLB on the PA-RISC [HH93], and refined further in the *clustered page table* [TK95] planned for an upcoming release of Sun Microsystem's Solaris operating system for the 64-bit UltraSPARC processors. An HRPT is a table of translations for physically resident pages, accessed by *hashing* on the virtual address. Hashed translation tables are efficient: in the common case (no collision in the hash table), the TLB miss handler can fetch a translation from the table with one or two data cache line references. A hashed translation table grows in proportion to the size of physical memory rather than virtual memory; the space overhead is insensitive to sparse address reference patterns. Because the HRPT is compact, it can be resident in physical memory; references to resident pages never incur a page fault on the translation table, as they can with hierarchical page tables, and TLB pollution by the address mapping structures is reduced.

The HRPT structure is derived from the "inverted page table" mapping structures used in IBM's 801 and RS/6000 segmented architectures, and in some global address space architectures, including the System/38 and Monads. Variants of this structure are highly successful in widespread commercial use. Although hashed tables are superior to tree-structured page tables for large, sparse address spaces, hashed tables present difficulties with the address aliasing needed for private address spaces. To my knowledge no private address space system currently uses a hashed translation table, except those implemented on segmented architectures, which map private virtual addresses through segment registers into an underlying global address space. However, HRPTs are quite natural for the single virtual address space structure proposed in this dissertation.

This discussion has skirted two important issues. First, although a HRPT can handle sparse translations for the *resident* pages of a large virtual memory, the operating system must also include mechanisms to locate nonresident pages and fetch them into memory. Section 3.2.1 addresses this issue. Second, the mapping structures in hardware and software must represent protection. Successful uses of the HRPT structure have been limited to architectures with specific hardware mechanisms for protection: segment registers on the IBM 801 and RS/6000, capability registers on the System/38 and Monads, and protection set registers on the PA-RISC. We must consider how to protect a global address space on a stock MMU designed to protect memory by private address spaces.

3.1.3 Software Protection Structures

On paged MMUs, any operating system supervisor enforces memory protection by ensuring that each processor's TLB contains no enabled translations for pages whose access should be denied to the protection domain of the code stream executing on that processor. The system must meet two conditions: (1) it must never load such a protected translation into the TLB, and (2) it must ensure that no such translations remain in the TLB after a protection context switch. This section considers the first condition, and Section 3.1.4 considers the second.

To meet the first condition, we can represent protection in the supervisor mapping structures by tagging each translation in the HRPT with one or more *protection keys* identifying the protection domain(s) that can access the named page. This is a variant of the HRPT for the PA-RISC, which includes a single protection key with each translation, granting access to any addressing context that "wields" a matching key in any of the four privileged protection set registers. On the MIPS and Alpha processors, we must assign a key value to each addressing context, and our HRPT structure must maintain keys with each translation to identify the contexts enabled to use that translation. The TLB miss handler checks for a key matching the current context before loading a translation into the TLB. This adds a few instructions to the TLB miss path, but no additional data cache line references.

The scheme for handling protection domain keys is straightforward. Keys are assigned sequentially as protection domains are created. The supervisor enters translations and keys into the HRPT as needed, for example, while handling TLB misses as pages are touched from within each addressing context. When a domain is destroyed, the system simply retires its key; its entries will naturally flow out of the HRPT with time. In contrast, setup and teardown of an addressing context can be an expensive operation in private virtual address space systems using hierarchical page tables. The tradeoff of this proposal is that the system must sweep the HRPT to cancel any stale keys before their values can be recycled, i.e., when the space of valid keys is exhausted. Note also that the HRPT can cache only a few keys for each page, since each HRPT entry should fit within a single data cache line to speed TLB miss handling. Thus there is a tradeoff in choosing the key size: smaller key values can improve sharing performance, while larger key values allow the system to delay key recycling longer.

3.1.4 Hardware Protection Structures

To meet the second condition for memory protection, translations in the TLB must be disabled on each protection context switch. On current implementations of the Alpha, the supervisor is forced to resort to a purge of all user translations in the TLB (and a purge of the first-level cache) on each context switch, regardless of whether the system uses private address spaces or a single address space. Context switches are cheaper (on average) on the MIPS R4000 and other processors in which each TLB entry includes a tag corresponding to the addressing context for which the entry was loaded. With these *tagged TLBs*, the supervisor can enable and disable specific groups of cached translations simply by changing a privileged processor register: the hardware validates the tag against protection register(s) on each TLB hit. On the

PA-RISC the tag is validated against the protection set registers; on the MIPS, the tag is matched against a single *address space identifier* (ASID) register. These tags are identical in concept to the "protection keys" in the previous section. A single address space system implements protection context switches in the same way as a private address space system, by replacing the value in the ASID register with the key for the newly activated addressing context.

Although tagged TLBs reduce the impact of protection context switches, it is worth noting that the ASID tags on the MIPS effectively prevent the active context from using any translations for *shared* pages left in the TLB by the inactive context. Thus the architecture may incur unnecessary TLB misses and possibly redundant translations for shared pages. This problem occurs whether or not a single virtual address space is used, but the single address space model offers an opportunity to improve the architecture by separating protection from translation in the TLB and other hardware caches. For example, the protection set keys on the PA-RISC permit processes sharing memory to use a common set of entries in the TLB and data cache. Also, a single address space eliminates address aliasing that can interfere with virtually indexed data caches. We have discussed these architectural issues in some detail in [KCE92].

3.1.5 Summary

This section has outlined hardware and kernel software issues for implementing a global address space with multiple protection domains on standard RISC processors. The key points of this section are:

- While the single address space structure is problematic on MMUs with architecturally defined tree-structured page tables, it can be implemented efficiently with a global hashed translation table (HRPT) on modern MMUs with software-loaded translation buffers.
- Hashed translation tables are well-understood and highly effective for representing large, sparse virtual memories. The performance of address translation today is dominated by the TLB hit rate, which is determined by the temporal locality of the reference stream, and is insensitive to the size or sparseness of the virtual address space.
- Protection domains within the global address space can be represented efficiently with protection key tags in the global software translation table and TLB. We expect that future versions of the Alpha and other wide-address processors will employ software-loaded tagged TLBs.

The picture is not yet complete. To build an effective single address space system, the supervisor must be able to efficiently handle misses in the global translation table. These include *protection misses* (requests from a particular protection domain to access a resident page for which no matching protection key is cached) as well as *page faults* (which require the supervisor to locate arbitrary pages in the global virtual memory and fetch them into local primary memory). These issues of location and access control are primary concerns for single address space systems. In the next section, we give an overview of how these and other concerns are handled in the Opal model.

3.2 Overview of the Opal System

In this section we outline the key elements of the Opal system model, with an emphasis on how Opal's abstractions are designed to simplify management of addressing and protection at both the system level and the application level, above the efficient but unstructured low-level model of a huge, flat, protected virtual address space. We focus on general concepts and their rationale, deferring a discussion of the specific primitives to Chapter 4.

The essential concepts are: (1) segments add structure to the monolithic global virtual memory, (2) capabilities control each protection domain's access to segments in a general way, and (3) an object-based programming model permits programmers to use protection domains and segments in order to create a protection structure that balances their needs for protection and performance. While the basic concepts of segments, capabilities, and objects are familiar from the systems discussed in the previous chapter, we emphasize that in Opal they are *logical* entities implemented in higher-level software above the global protected virtual memory.

3.2.1 Segmented Allocation and Sharing

Segments are the grain of virtual memory allocation, persistence of virtual address space, memory access control, and backing storage management in Opal. Segments amortize the memory management overhead across a collection of related pages and data items. At the application level, segments remove the need to interact with the system for each page of memory that is allocated or accessed; system primitives create segments of requested sizes in the address space, and obtain access to entire segments with a single operation. At the system level, segments simplify the translation and protection tables above the HRPT. Protection is maintained only as a list of segments accessible to each protection domain in most cases. Translation tables for a segment are shared by all users of that segment; these translation tables can be per-segment linear block maps (e.g., *inodes* [MJLF84]), extents, or any other structure appropriate for tracking the location of that segment's pages in secondary memory.

The inherent compromise is that segments are exposed at the system interface, and they reduce user control over sharing protection choices. In particular, applications must specify how data structures are partitioned across segments when they are created. Two data items placed in the same segment cannot be shared protected independently, yet it is prohibitive to place each item in a separate segment. The premise is that applications rarely if ever need to control access to individual items; rather, data items are grouped into connected data structures that are accessed as a unit.

Note, however, that addressing in Opal is not segmented in the sense of Multics and its derivatives (Section 2.2.4). In particular, segment boundaries are unknown to the hardware. All segments in an Opal system are simultaneously and directly visible in virtual memory (given proper authority) using fully qualified global virtual addresses. The virtual address of a segment is a permanent attribute fixed by the system at segment allocation time. The Opal protection model does not depend on nor is it compatible with segmentation hardware.

Because addresses are fully context-independent in Opal, data structures linked with native virtual address pointers can span segments, enabling different access controls on different parts of a connected structure. The availability of cross-segment pointers makes the constraint of segmented access control less onerous than in segmented architectures, in which segments are the unit of addressability as well as the unit of protection.

3.2.2 Software Capabilities

Opal uses capability-based naming and access control for segments, protection domains, and other system abstractions. Each system call to create a protected system resource (e.g., a segment) returns a capability conferring permission to operate on the newly created object. For example, a thread that allocates a virtual memory segment receives a capability for the segment; code executing in another protection domain cannot gain access to the segment unless it possesses this capability.

Capabilities allow the code that controls access to a resource to exist independently of the code that implements the resource. The *server-structured* operating systems that I am familiar with, including Microsoft NT, export all protected kernel services in terms of protected objects named by capabilities. Most of these systems (e.g., Mach [JR86], Amoeba [MT86], Chorus [RAA+88], and Spring [HK93]) also support user-defined protected objects as a basis for extending operating system services in a protected way. In these systems, user code with no inherent privilege can provide operating system services to clients who choose to “trust” them, as defined by the client’s willingness to hand over capabilities in order to benefit from the service. A client’s ability to obtain services is limited to the capabilities it receives from the server; a server’s ability to cause damage is limited to the capabilities it receives from clients. Untrusted entities cannot gain access, and untrusting entities are never adversely affected. Note, however, that the notion of "trust" is transitive, and capabilities can be stolen or mistakenly given away (this is the *confinement problem*).

For Opal, capability-based protection of segments allows multiple memory access control policies to coexist. Access privileges are determined independently for each component, not simply from the user identity on behalf of which the component is acting. This use of capabilities to "separate access control policy from mechanism" in extensible operating systems dates back to Hydra. It should not be confused with capability-based addressing (Section 2.3): the concept is the same, but the mechanism and granularity are different. The capabilities used in Opal and the other systems cited above are implemented entirely in software, as described in Section 5.2.3. Like virtual addresses, Opal capabilities are context-independent values that can be passed through protected procedure call or shared memory. However, capabilities are augmented with a large random number, making them probabilistically impossible to forge. We have chosen not to concern ourselves with the confinement problem: the software capabilities used in Opal can be revoked, and they may be augmented with additional access checks at the server's discretion.

3.2.3 Object-Based Programming Model

Facilities for sharing memory introduce new concerns for the system and its applications. In private address space systems, processes are the unit of software construction, program execution, and resource management. In Opal, programs are constructed from components active in different protection domains, and segment lifetimes are independent of protection domains. An alternative programming model can help application designers exploit the flexibility of the single address space.

The model should permit application designers to focus on components and their interfaces, rather than concerning themselves with the details of sharing and protection structures in their applications. If sharing exposes or affects internal details of component implementation, then it will be more difficult to implement, debug, and modify software that uses sharing. The details of managing sharing -- including resource control, cross-domain synchronization, code binding, and partitioning of data structures across segments -- should be isolated from the application code that is not directly concerned with setting up the protection structure. Our measure of success in achieving transparency is the ability to configure different protection relationships after components have been constructed, by adjusting protection boundaries between components to achieve the desired balance between protection and performance, with little or no impact on source code.

To meet this goal, we employ concepts from the object-based systems discussed in Section 2.2.3, to maintain an abstraction of a uniform space of objects in which protection boundaries are transparent. User data is structured as collections of heap-allocated objects linked by pointers; user code is packaged as procedures for particular object types. Threads navigate through the web of objects, following pointers and invoking methods. While these invocations are syntactically uniform, the semantics of each invocation are determined *by the access privileges of the caller* with respect to the invoked object. Local private objects are invoked with ordinary type-checked procedure calls; protected objects are invoked with protected procedure calls.

This approach is inspired by the "uniform object model" of Emerald [JLHB88], a distributed programming language with location-independent invocation of mobile objects. Emerald implements object invocations as local procedure calls when possible, but automatically resorts to RPC calls when the invoked object resides across the network. The Emerald runtime system implicitly selects the best implementation of each call, rather than explicitly programming the two styles into the application. In Opal, we use transparent object invocation to hide *protection* boundaries instead of or in addition to network boundaries. This is similar to Lipto [DPH92], except that objects in Opal may be directly shared by more than one protection domain, with local calls to a shared procedure operating on shared data. Shared memory and protected memory are viewed as *procedural* abstractions independent of modularity.

In operating systems and architectures with a single object model defined by the lowest levels of the system, objects were seen as synonymous with protection domains. Protection was seen as an essential aspect of modular structure, a view that survives behind some of the recent work with microkernel systems. In Opal, objects are lightweight data items that are independent of protection domains. Fundamentally, we believe that operating system protection structures are not the right level to impose modularity. In fact, protection structures do not impose modularity, they only *enforce* selected module (or object) boundaries. Object encapsulation can be enforced by language type rules where sufficient, and by hard protection boundaries where necessary. Protection structures should be flexible enough to permit application-specific choice of how modularity is enforced.

The object runtime facility described in Chapter 5 is able to achieve these goals in a reasonably general fashion, to demonstrate that objects can be used to structure sharing and protection interactions among protection domains in a general and flexible way, with minimal impact on the syntax or logical structure of the program. However, objects are not fundamental to the Opal system; they are merely a runtime system veneer on the underlying operating system facilities, which are unstructured and tuned for effiicient implementation on standard hardware, and could as easily support other programming environments that are not object-oriented. With that said, it is also true that the abstraction of a flat untyped global virtual memory is well-suited for object-based sharing systems, as discussed in Chapter 5.

3.2.4 Related Work

At this time there are several projects underway to build single address space operating systems similar to the Opal system. These projects include Mungi [HERV94], Nemesis [BMvdV93], and Angel [MWO+93]. These systems are similar to Opal in their essentials, emphasizing simplified sharing and persistence within a global 64-bit address space. Some known differences are outlined in later sections of this dissertation.

Opal and these other single address space projects owe a substantial debt to two preceding systems, Monads (Section 2.3.3) and Psyche [SLM90] at Rochester. Psyche was built to support *multi-model* parallel applications composed of cooperating components using different parallel programming models; these components are partially isolated in separate protection domains within a global paged address space. Psyche's *realms* enforce object-style access to data shared among models, primarily to ensure that model-dependent operations (e.g., acquiring a lock) are handled correctly regardless of which component invokes them.

Hemlock [GSB+93] extends the Psyche work to a Unix context, adding memory-mapped persistent storage and dynamic linking to shared code modules that encapsulate shared data. Hemlock is a hybrid global/private address space system: addresses at the low end of the processor address space have separate mappings in each process, with a large globally addressed region at the upper end. One goal of Hemlock is to preserve backward compatibility with Unix (Section 3.3.3) while supporting uniform addressing of shared memory.

3.2.5 Summary

This section gave an overview of the key elements of the Opal model discussed in the next few chapters. Opal's global virtual memory is partitioned into segments occupying disjoint regions of virtual address space. Opal provides system primitives to create segments and protection domains. Segments and protection domains are named by software capabilities that govern each domain's access to memory. The common address space and uniform segment sharing allow *dynamic* imports and binding to data structures and code. There is no need to decide in advance what is shared and what is private: any segment can be shared at any time without address conflicts, and segment access rights are easily passed on-the-fly from domain to domain. An object-based programming model permits cooperating components to interact through shared data and across protection boundaries without compromising the modular structure of the program.

3.3 Some Issues for Single Address Space Systems

The flexibility of the Opal model arises, in part, because virtual addresses have an unambiguous global interpretation in the single address space. In this section we discuss several tradeoffs that are inherent in the pure single address space approach. In general, these tradeoffs are related to the inability to use context-dependent addressing in a pure single address space system. That is, private address space systems may benefit from the assignment of different meanings to the same address, but this choice is not available in a single address space system. Also, the system must treat virtual address space as a global system resource. As with other shared hardware resources (e.g., disk space and physical memory), the system must ensure that address space is used fairly and that the legitimate needs of all users are met. This requires accounting and consumption limits, and the system must deny any request that cannot be satisfied from the available address space.

These restrictions are not visible in practice if the hardware address space is large enough to meet legitimate needs. We have not answered the question of how much address space is “enough”; our goal is to define and evaluate the single address space structure on the assumption that wide-address processors will soon provide sufficient virtual address space for this approach, without constraining applications.

3.3.1 Virtual Contiguity and Segment Growth

One problem with the single address space structure is that the ability of virtual segments to grow cannot be guaranteed. If a segment turns out to be too small because its needs could not be anticipated in advance, then the application must allocate another segment. (Of course, this is true of segmented systems as well.) This is acceptable for heap-allocated linked data structures, since connected data structures can easily span segments in a single address space system. We considered adding a *segment group* mechanism to permit a collection of related segments to be viewed as a unit, but this idea was not developed further.

Even if segment groups are supported, segments allocated at different times will not be contiguous in the address space. The loss of contiguity is not generally a problem for programs written in high-level languages, but it limits the use of indexed memory structures (e.g., arrays) that must be stored contiguously. Programs must allocate segments that are large enough to store these structures in advance.

Because the virtual address space is finite, the system must impose limits on the amount of address space that can be preallocated to allow contiguous structures to grow. Of course, maximum segment size is limited on current 32-bit systems as well; in fact most current operating systems limit the maximum virtual space to a fraction of their four-gigabyte architectural limit, which itself is less than one billionth of a full 64-bit address space. A single address space system can thus easily provide larger segments than today’s private address space systems, but not as large as a private address space system on 64-bit hardware. We emphasize that the single address space approach reduces the address space available to individual applications; it is useful only if there is "sufficient" address space to render the issue moot.

3.3.2 Memory Reclamation and Address Recycling

In private address space systems it is a simple matter to reclaim the memory resources of a terminating process for use by other processes; the system simply destroys the address map for the process, and reallocates the physical memory used or referenced by the map. In a single address space, reclamation is no longer tied to program entry and exit, or even the creation and destruction of protection domains.

In the absence of sharing and persistence, managing a segmented global address space is not particularly difficult. Opal uses segment-grained memory reclamation of virtual and physical memory at the system level. By default segments are reclaimed when there are no protection domains with permission to access them, i.e., when the last entity using their data is destroyed. This simple policy is supplemented with controlled reference-counting primitives for long-lived segments. The *resource groups* described in Section 4.5.1 support accounting and resource control for segments and other resources, independently of the protection domains that can access those resources.

Given finite virtual address space, the system must eventually recycle address ranges that have been used and released. This raises the question of how dangling references are handled. Dangling references can occur in any system that supports a shared name space but does not provide global garbage collection. With Opal, we are unwilling to enforce a global data model needed for garbage collection, although we contend that garbage collection could be provided within a particular programming environment above Opal using established techniques above the primitives presented in Chapter 4. Nevertheless, dangling references can occur across environments.

Address recycling can occur at several levels. For example, heap managers recycle heap slots within a segment to conserve physical memory as well as virtual address space. In this case, dangling references cannot affect other entities that are not using that heap segment. Should the operating system reclaim the segment and later reassign it, however, an entity with retained references into the original segment might then erroneously attempt to use a pointer to the reassigned segment. This class of dangling references -- like all erroneous address references -- must be handled by the access control mechanism. That is, neither dangling nor erroneously generated references will permit access to memory, unless that memory lies within a segment attached to the current protection domain.

Opal avoids recycling across segments as long as it can; if the address space is ‘‘large enough’’ then addresses will never be recycled. Crow and Kotz [KC94] compare several address space allocation and recycling policies for Opal and other single address space systems.

3.3.3 Process Creation by Cloning Address Spaces (Fork)

Private address space systems use address space creation primitives as a means of executing programs. In Unix-like systems, address space creation (with the **fork** primitive) replicates the parent process context, including its private virtual memory, in a child process. A pure single address space system cannot support aliasing and therefore cannot emulate any form of address space creation primitive. In short, Opal cannot support Unix, although Unix can coexist with Opal above a general-purpose microkernel (Chapter 6).

In the past, there were two reasons to clone a parent address space: (1) to create multiple concurrent processes executing the same program, and (2) to allow code from the parent to initialize state (e.g., file descriptors) in the child. Lightweight threads are a better primitive for concurrency in modern systems, and the system can provide a means to initialize a child without executing code in the child’s context. In fact, **fork** is a source of complexity and inefficiency in Unix systems: the majority of the state copied by a fork is not typically needed by the child, inspiring efforts to improve performance by deferring the copy (e.g., with copy-on-write) or avoid it entirely (e.g., the **vfork** primitive in early BSD Unix). The copying semantics of **fork** also interfere with support for threads [MS87].

Opal replaces **fork** with primitives to create protection domains and initialize them by attaching code and data segments and installing RPC endpoints. Threads in the parent domain can then make cross-domain calls to enter the child domain; for example, the object support package above Opal allows a component to “execute a program” by creating a protected object and invoking it across a protection boundary; Spring apparently provides a similar facility. Finally, although compatibility with Unix is not our goal, most Unix programs do not use **fork** directly, and could run in a single address space environment.

3.3.4 Address Remapping and Copy-on-Write

One objective of the single address space approach is to discourage applications from sharing data by copying it between private virtual memories, and replace copying with by-reference sharing wherever possible. Obviously, not all instances of copying can be eliminated. In particular, data may be copied to create a new version to be modified independently of the original, either for protection reasons or to preserve the original.

When data is copied within an address space, some pointers must be converted to point into the new copy. Conventional systems can sometimes avoid this conversion when data is copied from one process address space to another, by placing the copied data into the same virtual address range in the receiver process as it occupied in the sender. (Of course, it is then impossible for either process to name both versions.) Since single address space systems have only one naming context, the copy must occupy different virtual addresses than the original, and pointers must be handled specially. Furthermore, a conventional system can lazily evaluate such a copy using copy-on-write, while a single address space system cannot. In the single address space system, pointers in the data must be relocated before the receiver can even *read* the data (otherwise, the receiver might point to the wrong version when the copy is made); copy-on-reference could be used, but not copy-on-write.

The most important uses of copy-on-write stem from the **fork** primitive, which is useful only for compatibility with Unix. However, the absence of remapped copying and address overloading causes several problems for the handling of code segments in our Opal prototype (Section 7.3). Opal must copy initial values of writable static data when a code segment is loaded (instantiated), and internal pointers to static data must be relocated at load time. Programs written using our object package generally contain minimal writable static data, but this cost will be higher for certain programming styles. Also, it is difficult to convert shared data to private. For example, current systems overload virtual addresses when shared code is modified, e.g., to set a breakpoint.

3.3.5 Discussion

This section has outlined some issues raised by the single address space approach. Fundamentally, these issues stem from (1) operating system control of the virtual address space as a shared system-wide resource, and (2) the inability of the pure single address space model to support context-dependent addressing. We have argued that these tradeoffs are acceptable given sufficient hardware address space. In particular, context-dependent addressing is sometimes useful, but an equivalent or better alternatives exist in most cases. Furthermore, context-dependent *naming* may be useful in some situations, but context-dependence can appear at higher levels of naming interpreted by software (e.g., relative to a thread's registers) and mapped into a single context-independent address space interpreted by the machine.

3.4 Summary

This chapter outlined the single address space model as embodied in the Opal system, and its implementation on standard RISC processors with paged MMUs. The objective of the single address space operating system is to expand the choices in the structuring of computations, the use of protection, and the sharing, storage, and communication of data, by guaranteeing uniform, context-independent virtual addressing across all entities in the system. Opal combines the positive features of earlier systems in a unique way, unified by the single address space on 64-bit architectures. We now turn to a more detailed description of the Opal system interface.

Chapter 4

The Opal System Interface

This chapter presents the basic Opal system abstractions and primitives. We focus on the low-level facilities that control the interactions among protected application components, including the primitives for managing allocation, sharing, and reclamation of the global virtual memory. This interface is suitable for implementation as a native kernel or as an Opal subsystem server above a standard microkernel, as described in Chapter 6. While these kernel primitives determine the addressing and protection model exported to user software, they do not by themselves define the end user's or application programmer's view of the system. We must consider them in the broader context of additional system software at several levels:

- **Runtime systems** define programming interfaces that are intended to match the needs of particular programming languages. For example, runtime services in Opal support concurrency, synchronization, and object-based protection and communication for programmers using the C++ language (see Chapter 5).
- **System utilities** including shells, editors, window managers, etc., provide the user interface and tools to program the system. Opal need not have any impact on user interfaces, but there are important implementation differences for linking and debugging utilities, discussed in Chapter 7.
- **Protected servers** or subsystems support application-level facilities to mediate the interactions of mutually distrustful clients. Section 4.6 outlines several services for symbolic naming, data management, and access control within the Opal environment.

The division of function among these levels is not merely an implementation detail; it is fundamental to the structure of Opal or any system. In particular, it determines the ways in which the system can be extended. Runtime systems, utilities, and servers are *replaceable* components: they can be tailored to application needs, and different versions can coexist in the same system. The purpose of this chapter is to define the essential system features that cannot be replaced. These facilities are designed to be general and efficient, but higher-level support is essential to make them easy to program. I include details of runtime systems and higher-level services when useful to simplify the presentation and to illustrate how the kernel primitives can be used.

All aspects of the Opal system are designed to support flexible protection and sharing within the single virtual address space. Within this broad framework, the basic facilities that Opal must support are the same as for any operating system. These facilities are listed below, along with summaries of the corresponding Opal kernel abstractions. Since the basic needs are the same, these abstractions are similar and in some cases identical to those of many other systems, with certain crucial differences.

- **Data Storage**. Opal's global virtual memory is partitioned into *segments*, sequences of virtual pages occupying a contiguous range of virtual address space. All data in virtual memory (including executable code) resides in some segment.
- **Execution**. *Threads* are the active agents that execute user code and operate on user data. A thread is an executing stream of instructions and its procedure call stack. An application may execute as multiple threads.
- **Protection**. Threads execute within *protection domains* that limit their access to a specific subset of segments in the global virtual memory with particular modes (e.g., read-only or read-write privilege). Opal domains are passive protection contexts; alternatively, the protection domain can be viewed as the collection of segments accessible (*attached*) to such a protection context.
- **Communication**. Opal supports protected control switches across protection domain boundaries through numbered entry points called *portals*. Control transfer through a portal is the only means to execute code within a protection domain.
- **Resource Control**. All actions that consume system resources (e.g., creating virtual memory segments) are associated with *resource groups*, representing users and the software entities acting on their behalf. The resources "charged" to a resource group may be limited by the system according to a resource control policy. Also, user software can release the resources in a group as a unit.

These five basic abstractions -- segments, threads, protection domains, portals, and resource groups -- are system objects created and destroyed by explicit calls from user software. Protected system objects for segments, protection domains, and resource groups are named and accessed using unforgeable identifiers -- capabilities -- that can be passed and shared among user software entities. The implementation of these capabilities is described in Section 5.2.3.

The following sections describe the basic abstractions in more detail. To streamline the presentation I make two simplifying assumptions. First, I present idealized kernel interfaces as they appear to users of Opal's standard runtime package, which maintains typed records (descriptors or *proxies*, Section 5.2) for protected system objects, containing the capability for the object and other hidden status information, such as the first and last addresses of a segment. Second, I assume that the target hardware architecture is a single node with one or more homogeneous processors and a shared physical memory, and that all resources are transient. The global address space can be extended to include data in long-term storage and across a network, as explained in Chapter 8, but the concepts and abstractions described here are independent of persistence and distribution, and the discussion avoids these issues.

4.1 Storage and Addressing

User threads allocate segments of virtual memory with the *NewSegment* primitive. The system assigns a fixed virtual address range to each segment when it is created; this range is disjoint from the ranges assigned to all other segments. A newly created segment is made accessible to the creating thread's protection domain in read-write mode, allowing the thread to access the segment's pages directly with **load** and **store** instructions. Physical memory for the segment is allocated zero-fill on demand as the segment's pages are referenced; backing storage is allocated as needed to hold pages that overflow physical memory. Some have argued against lazy allocation on the grounds that it will fail unexpectedly if inadequate physical storage (e.g., swap space) is available for a page when needed; this problem is easily avoided, where it exists, by adding a primitive to preallocate storage and bind it to a region of address space.

NewSegment returns a capability that can be used to grant other entities permission to access the newly created segment. This capability names the segment indirectly through a "segment reference" or *SegmentRef*, as shown in Table 4-1. Segment references are explained in Section 4.5.2; briefly, they support (1) referencing of the same segment through multiple capabilities, possibly with different access privileges, and (2) protected reference counting of shared segments. I emphasize that segment capabilities and segment references exist only to *set up* protected sharing of segments. Once access to a segment is established, each thread addresses the segment's contents directly at its assigned virtual addresses, independent of the thread's location in the system.

A segment's internal structure is known only to the application software that uses the segment. Each segment typically contains a package of procedures or a self-contained heap for allocating complex data structures. Opal segments are *logical* clustering units for related user data, allowing preallocation of address space, and one-step binding, location, and access control for a collection of related data items.

Table 4-1: Basic Segment Primitives.

NewSegment (**ByteCount** length, **Boolean** aligned) returns **SegmentRef**. *Create a segment of a requested size, rounded up to a system-determined alignment boundary (at minimum a page boundary).*

SegmentRef.FirstAddress () and **SegmentRef**.LastAddress () returns **Address**. *Runtime call to retrieve the first and last address of a segment.*

SegmentRef.WriteEnabled () returns **Boolean**. *Runtime call to determine access allowed to the segment using this* ***SegmentRef****.*

4.2 Execution and Concurrency

Opal provides conventional primitives for creating and managing threads of control, shown in Table 4-2. Threads are implemented in a concurrency library called OThreads [FCL93] above raw kernel primitives for processor allocation. Opal and OThreads were conceived to use *scheduler activations* [ABLL91] to animate threads, but the details of these kernel primitives are independent of other aspects of the Opal system and are beyond the scope of this dissertation. Without loss of generality, I will speak of threads as if they are first-class system objects.

A thread is an independently schedulable execution entitity, consisting of a descriptor and a procedure call stack in memory. The descriptor contains a set of register values, including a program counter, and other thread-specific state. The runtime system scheduler executes a thread by loading its register values into a processor; it suspends a thread (e.g., for blocking synchronization or timeslicing) by gaining control of the thread's processor and writing its register values back to its descriptor in memory. A suspended thread can be resumed later, possibly on a different processor.

Threads are multiplexed on processors, and different threads may execute in parallel on different processors. If multiple threads operate on the same area of memory then there is logical concurrency, and possibly physical concurrency if the node has multiple processors. Shared procedures must be reentrant. With OThreads, the application manages synchronization and mutual exclusion with standard mutexes, condition variables, and spin locks.

Threads in Opal are independent of any addressing context. The instructions a thread executes and the memory it references are determined entirely by the values in its registers (e.g., the global virtual addresses of its stack pointer and program counter), the state of its descriptor, and the value it sees in the global virtual memory. For example, the runtime system satisfies all heap allocation requests made by the thread from a "current heap" region (or *memory pool*, Section 5.1.1) for the thread, designated by a pointer in its thread descriptor. The current heap can be changed with runtime calls. A thread's descriptor and stack are allocated from the current heap region of the thread that creates it; the current heap and scheduler are also inherited from the parent thread.

Table 4-2: Thread Primitives.

NewThread (**Address** procedure, **Untyped** argument) returns **Thread**. *Create a thread of control and start it in an entry procedure.*

Thread.Join () returns **Untyped** result. *Block until a thread returns from its entry procedure, then return the value produced by that procedure.*

4.3 Memory Protection

Opal provides *protection domains* to restrict memory access privileges, in order to protect sensitive data from access by unauthorized threads. A protection domain is a passive execution context for threads, limiting their access to a particular set of segments. Any thread may execute a **load**, **store**, or **branch** instruction to any part of the global virtual memory, but the reference will not complete unless the segment containing the addressed page is accessible to the thread's protection domain in the required mode (**read**, **write**, and/or **execute**).

Domains rather than threads are the *subjects* of memory access control; they are grouping units for teams of threads with common memory access privileges. Every thread executes within some protection domain, although a given thread could be resumed in different domains at different times. In the simplest case, all of the threads of an executing application run within the same protection domain. Segments are the *objects* of memory access control. Segments are protected from code by excluding them from the protection domains in which that code executes; domains are protected by limiting the threads that can execute within them.

At the lowest level, user code uses *Attach* and *Detach* primitives to establish and revoke a protection domain's access to a segment. *Attach* is the Opal analogue of the *mmap* file mapping primitive in Unix-like systems, or the *load segment* operation in segmented systems (Section 2.2.4). Like those primitives, *Attach* must locate the segment, authorize local access to it, and set up internal system tables to permit the program to address the segment directly. While the setup may be expensive, involving disk activity, network communication, and updating of protected system tables, subsequent references to the segment are very efficient. *Attach* differs from *mmap* in two critical respects: (1) the system rather than the user program selects the address for the attached segment, and (2) *all* data in Opal resides in segments that are potentially attachable; no memory is inherently "anonymous" or private to any particular protection domain or process.

Table 4-3: Protection Domain Primitives.

NewDomain () returns **Domain**. *Create a protection context with no memory access privileges.*

Domain.Attach (**SegmentRef** segment). *Make the specified segment accessible to a domain's threads,with the modes allowed by the* ***SegmentRef****.*

Domain.AttachReadonly (**SegmentRef** segment). *Make the segment named by a* ***SegmentRef*** *accessible to a domain's threads, but disallow store operations.*

Domain.Detach (**SegmentRef** segment). *Remove a segment from a protection domain, revoking access to the segment by any threads in the domain.*

Domain.Destroy (). *Destroy a protection domain, ceasing execution of all active threads within the domain.*

4.4 Interdomain Communication with Portals

Opal provides a simple mechanism called *portals* to change the protection context of the processor. A portal is an entry point for control transfer into a protection domain, uniquely identified by an integer value (the *portal ID*) assigned by the system when the portal is created. Portals are the basis for protected procedure call in Opal, and the sole means to execute code within a newly created protection domain.

Portals are protected in the following sense. First, only a holder of a domain capability can create a portal for the domain. Second, threads entering a portal begin executing at a fixed virtual address chosen by the creator of the portal. These are the key properties that permit the holder of a domain capability to exclusively control what code is executed within that protection domain. However, once created, no authorization is needed to gain initial entry through a portal; any thread that knows the value of a portalID can transfer control into the portal's domain. Ordinarily, a portal is created for a collection of procedures, and the code executed on entrance to the portal is a server-side dispatcher stub that validates arguments and checks authority of the caller. Uniformly named portals are fundamental to the implementation of capabilities in the Opal/C++ runtime package, described in Chapter 5, which adds key-based access control checks on the receiving side of a portal.

Portals were designed as a minimal basis for protected procedure call, trap delivery, and kernel upcalls to user code, generalizing the *Processor.Donate* primitive that underlies Bershad and Anderson's work with high-performance communication and scheduler activations [BALL90]. Portals can support a range of cross-domain call semantics involving different user-level conventions for register usage, stack switching, response handling, transmission of arguments and results through shared memory, and authentication. The global name space for portals allows exchange of cross-domain call bindings (e.g., capabilities) through shared memory, and permits uniform referencing of protected entry points through identifiers separated from the actual source and target of the communication.

Table 4-4: Portal Primitives.

Domain.RegisterPortal (**Address** entryPC) returns **PortalID**. *Create an entry point for a protection domain, named with a unique identifier.*

PortalSwitch(**PortalID** pid). *Switch the processor to the protection domain associated with the portal, and branch to the portal's entry point. Leave registers other than the PC unchanged.*

Domain.RebindPortal (**PortalID** pid, **Domain** delegant, **Address** entryPC). *Change a portal's receiving domain and/or its entry point.*

Domain.UnregisterPortal (**PortalID** pid). *Disable a portal; subsequent attempts to switch through the portal do not complete*.

Cross-domain control transfer through portals is the key to flexible memory protection at runtime. Logically, threads move between passive protection domains in a controlled way (with protected RPC) to change their memory access privileges. Protection domains are created to restrict or amplify the memory access privileges of particular procedures or application components; portals are used to switch to the chosen protection domain to execute the procedures. In effect, every protection domain is an RPC server. New domains are created as idle RPC servers; their parents (or peers) activate them with RPC calls. For example, a thread running in one protection domain (the *parent* domain) can create a new (more restricted) domain (a *child*), typically to protect the parent's data from an untrusted subprogram. The parent registers a portal for the child, specifying a procedure as the entry point, and then transfers through that portal. Parents can attach arbitrary segments to their children and cause arbitrary code to execute in their children. Parents may also pass portal identifiers to other child domains, effectively setting up RPC connections among siblings. (A "child" domain can also be created to amplify the parent's rights in a protected way, but in this case a privileged server must create the child on behalf of the nominal parent.)

4.5 Resource Management and Reclamation

An operating system must ensure that hardware resources are shared fairly among competing users and programs. To do this, the system must have some means to account for resource consumption and to reclaim idle resources for recycling. Opal's single virtual address space raises new concerns for this issue of *resource management*. Most obviously, virtual address space in Opal is a shared system-wide resource that must be managed as other hardware resources. Even supposing infinite virtual address space, physical storage named through the virtual address space must be controlled. Moreover, capabilities and virtual addresses are easily passed and shared, thus the lifetimes of the referents -- segments and other resources -- must be independent of protection domains, threads, or processes.

A fundamental principle of Opal is that all aspects of the application data model, including fine-grained resource management, are left to user-mode software. This design choice was made for reasons of generality and performance, as explained in Section 4.5.3. It implies that the Opal kernel does not track or control the placement of capabilities and addresses, which can be freely copied and shared in memory. Therefore, the kernel cannot by itself determine when to reclaim a particular resource, such as a virtual memory segment. Instead, our objective at the operating system level is to provide a general interface for applications and support facilities (runtime systems, language implementations, and garbage collectors) to manage memory and other resources for themselves, while allowing user software to protect itself from reckless resource handling by other entities. The resource management interface incorporates three features to meet this objective:

- Allocated resources are associated with protected system objects called *resource groups*, allowing the system to "charge" users and their software for the resources they consume. Since the system can identify all resource consumers, it can track and possibly limit their consumption with resource control policies, e.g., quotas or usage charges.
- User software can always release the system resources for which it is charged, using explicit release primitives. For example, a protection domain can be destroyed given a capability. All resources are reachable through their resource groups; anonymous resources are never "lost".
- For segments, which are shared in complex ways, resource ownership is generalized to explicit reference counts. *Reference objects* protect the resource references on a shared resource originating from different consumers, to prevent any consumer from prematurely releasing resources that are still in use by some other entity.

In short, the basic primitives allow explicit release of references to logical operating system resources, and are intended for use by application-level resource managers. Resource groups and reference objects exist to deal with erroneous or malicious managers who may release their resources prematurely, or not release them at all. These mechanisms are described in the following subsections.

4.5.1 Resource Groups

Opal defines *resource groups* as the identities that "own" logical system resources, including segments and protection domains. Every call to create a resource or to increment a reference count must pass a capability for a resource group as an argument. The standard runtime system associates a *current resource group* with each thread, and passes a capability for that group as a hidden argument for certain system calls (thus resource group arguments are omitted from the interface signature tables in this chapter). Multiple threads may share the same resource group. A thread can change its current resource group with a runtime system call.

Resource groups are the basis for accounting and resource control. Initially, there is one resource group for each user (or other accountable entity), though new groups can be created, as explained below. The system may choose to deny a request to allocate a resource for a particular resource group. In addition, resource groups can be used to forcibly release some or all resources allocated by a particular software entity. This is useful as a backup when higher-level reclamation mechanisms are not trusted, or if a software entity fails.

Resource groups are nested to allow a finer grain of control over accounting and resource management. Any holder of a resource group capability can create and delete *subordinate* resource groups, or *subgroups*. Thus the set of resource groups is a collection of trees. Accounting charges flow up the tree: resources allocated by a subgroup are charged to the parent, and transitively to a user (represented by the root of the tree) who is ultimately responsible for resource consumption by software acting on his behalf. Deletion privileges extend down the tree: the holder of a resource group capability can delete any descendent resource group, releasing all resources in that group, and so on.

In some respects, resource groups are similar to the hierarchical directories in Unix-like file systems. With either mechanism, an entity's permission to allocate resources (e.g., files or segments) is limited to specific directories or groups, which can be examined in a standard way and reclaimed as a unit. Like directories, resource groups can be assigned symbolic names to permit browsing through the space of allocated resources. However, with resource groups, a collection of related anonymous resources of different types may be grouped under a single name. Also, resource groups are generally memory-based rather than disk-based.

The important point is that resource groups are accounting identities that are decoupled from users, threads, and protection domains. This permits flexible accounting of resource usage in a highly dynamic system. Threads within a single protection domain may allocate resources under multiple resource groups; for example, threads within a protected server may use different resource groups to charge the resources they allocate on behalf of different clients. Threads in different domains may use the same resource group; for example, a single application entity using multiple protection domains internally may choose to account for all of its resources together. Subgroups support accounting of different software entities at an arbitrarily fine grain. For example, a thread may create a subgroup to track resources allocated on its behalf by an entity it cannot control, such as a server. A common use of resource groups is to create a subgroup for use by an untrusted procedure and destroy it when the procedure returns, to preclude any lingering side effects. In this way the traditional policy of releasing resources held by a terminating process is easily emulated.

Table 4-5: Resource Group Primitives.

NewResourceGroup (**ResourceGroup** parent) returns **ResourceGroup**. *Create a new resource group subsidiary to the designated parent.*

Thread.PushResourceGroup (**ResourceGroup** group). *Designate a resource group as the "current group" for a thread; all system resources allocated by a thread are entered into its group.*

Thread.PopResourceGroup (). *Restore a thread's current resource group to its value at the time of the last **PushResourceGroup** call for that thread.*

ResourceGroup.Destroy (**ResourceGroup** parent). *Destroy a group and all resources (or references) associated with it.*

4.5.2 Reference Counting with Reference Objects

While some resources (e.g., protection domains and resource groups) are reclaimed with explicit delete primitives, segments are managed with the reference counting primitives listed in Table 4-6. These primitives are designed to allow safe and flexible reclamation of shared resources by isolating the references registered by different entities. While segments are the only resources with fully general user-directed reference counting in the current Opal interface, the principles could just as easily apply to other resources.

Reference counting is implicit and automatic for segments that exist only as long as they are actively shared. An attached segment has an implicit reference registered to the resource group of the domain to which it is attached; by default, segments disappear only after the last detach. The reference counting primitives in Table 4-6 are used to preserve *unattached* segments, for example, segments registered with a name server (Section 4.6) or containing long-lived data (Chapter 8). Reference counts managed by these primitives are *explicit*. In particular, the reference count for a segment does not necessarily reflect the number of capabilities for that segment, or the number of virtual addresses pointing into the segment. Instead, the reference count indicates the number of entities that have explicitly registered an interest in the segment; later, those entities or others must issue explicit calls to decrement the reference count.

While resource groups track the references registered by each software entity, *reference objects* separate the counts to a shared resource emanating from different entities. This prevents a non-trusted thread from releasing more counts than it requested, triggering premature deletion of a shared resource. Reference objects are internal to the implementation of a resource, but their use is reflected in the interface to the resource. For example, each segment capability names a segment indirectly through a reference object called a *SegmentRef*. New *SegmentRefs* may be "cloned" for a given segment, each with a private reference count, and each named by a separate capability. Any thread with a capability for any *SegmentRef* may attach the underlying segment, but it can modify the counts for only the *SegmentRef* named by its capability.

Table 4-6: Segment control using *SegmentRef* reference objects.

SegmentRef.Reference(). *Register a reference to a segment by incrementing the reference count on a **SegmentRef**. If this is the first reference to the **SegmentRef** then enter it to the calling thread's resource group.*

SegmentRef.Clone () returns **SegmentRef**. *Create a new reference object for a segment, with the same resource group and access rights as the parent reference.*

SegmentRef.CloneReadOnly() returns **SegmentRef**. *Create a new reference object that confers read access to the segment, but not write access.*

SegmentRef.Release (). *Release a reference by decrementing the reference count on a **SegmentRef.** Destroy the **SegmentRef** if the count transitions to zero.*

In addition to protecting reference counts, reference objects help manage segment sharing in three ways. First, charges for a shared segment may be spread across multiple resource groups. Each *SegmentRef* is charged to the resource group of the thread that registers the first reference to it. (A newly cloned child *SegmentRef* is unreferenced; it is invalidated if its parent is reclaimed before the child is referenced.) Second, reference objects support restricted access: for example, a cloned *SegmentRef* can be "refined" to confer read-only permission with *CloneReadOnly* in Table 4-6. Third, reference objects support selective revocation: for example, a *SegmentRef* can be invalidated independently of others for the same segment.

Segment persistence is controlled with *persistent* reference counts, which are unchanged by a system restart. Persistent *SegmentRef* counts are maintained independently of the counts described, which are reset by a system restart. Any resource group containing a persistent segment resource is also persistent. In addition, all symbolic name entries (Section 4.6) are persistent, and hold a passive count on their referents, charged to the resource group that owns the name entry. Persistence raises a large number of issues, discussed in Chapter 8.

4.5.3 Discussion: Managing the Global Address Space

Any system with uniform naming must manage the sharing that the common name space is intended to encourage. For Opal, the key problem is that the system cannot know when to reclaim virtual memory unless it tracks pointers and capabilities in the global virtual address space, which can be viewed as an arbitrarily connected pointer structure.

Many systems that support uniform referencing have some means of controlling the flow of references through the system. Capability-based architectures (Section 2.3) track references directly in hardware, for example, by tagging memory locations that contain pointers, and using specific instructions for pointer manipulation. In single-language sharing systems (Section 2.2.1), including Cedar, Pilot, and Emerald, reclamation relies on the strong language type system, which restricts pointer arithmetic to control the creation of pointers, and can determine the location of pointers from the types of data records in memory. These systems can safely reclaim memory with global garbage collection.

We are unwilling to impose a single language or a global data model on applications at the operating system level. This is not to say that garbage collection is not needed. Rather, the issue is what *level* of the system should impose the constraints needed for garbage collection, and at what granularity. Opal must run on hardware systems that do not support pointer tracking, and the system must safely support multiple programming environments with different approaches to automatic garbage collection, including no collection at all for popular languages that are not strongly typed. Furthermore, it must be possible to pass references without operating system intervention for reference tracking in most cases. For these reasons, Opal delegates fine-grained (record-level) memory management to applications and language environments.

Opal's kernel facilities are designed to safely support multiple language-level memory managers within the single address space. The system allocates and reclaims name space in coarse blocks (segments); the managers allocate language records from within the name space of some set of segments, track references as necessary, and use the release primitives to return discarded segments to the operating system. Of course, each manager depends on certain application conventions: (1) it must know the format of data structures within its segments, (2) it must control or be aware of external references into its segments, and (3) it may assume some control over the threads operating on its segments. Each manager uses segment access control and protection domains to prevent interference from threads that are not trusted to adhere to the usage conventions, thus creating a safe sphere of influence. At the same time, resource groups exist to encourage proper memory management by identifying and controlling entities that consume resources excessively.

System-level memory management with segments and resource groups is safe, simple, efficient, and flexible. It supports automatic garbage collection for applications that voluntarily accept the costs and constraints, and it isolates each manager from damage by other managers. However, segment-grained memory management may worsen the problem of *internal fragmentation* within the single address space. The concern is that applications will preallocate large segments for their data structures, then be forced to retain segments that have only a few useful records remaining within them. The problem can be solved by heap compaction within each manager, but if virtual address pointers are used then the manager must update all pointers to the objects it moves. We anticipate that most applications will retain data structures for a time and then destroy them as a unit; if each segment is used only for related data structures, then fragmentation should not be a serious concern. Long-lived, evolving data structures must rely on runtime-level heap management, including heap block recycling and possibly compaction, to ensure good paging performance as well as to keep virtual and physical storage consumption to a reasonable level.

Opal's kernel facilities do not prevent individual managers from deleting objects prematurely, causing *dangling references* of various forms. Dangling capabilities are probabilistically detectable because they contain a randomized 64-bit check field (Section 5.2.3). At the system level, dangling virtual addresses are viewed as an access control problem. Programs granted access to a shared segment are trusted to use it correctly; erroneous pointer references always result from incorrect use either by the thread that stored the pointer, the thread that followed the pointer, or the thread that deleted the pointer's target. User code is responsible for using the available mechanisms (including protection domains and segment access control) to protect itself from damage caused by failures of untrusted threads.

4.6 Naming and Access Control

Capability-based segment access control is general and flexible. Capabilities can be freely shared, and any thread wielding a capability for a protection domain and a segment may attach the segment to the protection domain. Of course, the system must provide some means to distribute capabilities in a controlled way that can be used directly by applications, for example, to set up their initial interactions. We argued in Section 3.2.2 that user-supplied services may act as intermediaries that index and distribute capabilities to meet the needs of specific applications. Our Opal prototype includes several simple facilities to offer or obtain access to segments, outlined in Table 4-6.

- *Symbolic naming.* A simple *name service* supports a flat symbolic name space for segment capabilities. The name service can be augmented with an *access control list* (ACL). The current name server interface has a trivial ACL facility: access to a name entry is optionally limited to a particular resource group.
- *Attach-on-demand.* Access to a segment may be requested by virtual address, using the *AttachByAddress* call. This service can be used by the runtime system to transparently attach segments on address faults, as an application choice. A segment owner must *Publish* a segment capability before a client can attach the segment by virtual address.

- *Producer-consumer integration*. Section 6.4 describes a simple framework for transparently arranging sharing interactions among components that are not aware of each other's existence. Clients register with an *event service* as producers or consumers of named data structures. Producers permit readonly access to the segments containing their structures, which are attached to consumers automatically by the event server. Producers implicitly notify consumers of updates to the structures by reporting *events* to the server, which propagates them to consumers.

All of these facilities are replaceable system services, intended as examples of layered support for organizing and controlling access to data in the global address space.

Table 4-7: Distributing segment capabilities by symbolic names and addresses.

NameService.Enter (**String** name, **SegmentRef** seg, **ResourceGroup** principal). *Register a symbolic name for a segment capability in the system name server.*

NameService.Lookup (**String** name, **ResourceGroup** principal) returns **SegmentRef**. *Retrieve a segment capability by its symbolic name.*

NameService.AttachByName (**String** name, **Domain** d) returns **Address**. *Attach a segment to a domain, using a symbolic name for a segment reference.*

SegmentRef.Publish (**ResourceGroup** principal). *Permit a segment capability to be looked up by virtual address,using **AttachByAddress**.*

Domain.AttachByAddress (**Address** a, **ResourceGroup** principal). *Attach a published **SegmentRef** given a virtual address within the segment.*

4.7 Summary

This chapter described the basic Opal kernel abstractions: protection domains, segments, portals, and resource groups. These kernel abstractions are extended with application-level facilities, such as name servers, memory managers, and runtime support for threads, to produce the programming interface seen by applications. The kernel defines general mechanisms for managing system objects, including segmented access control and protected reference counts, in order to safely support a range of application-level facilities for naming, access control, and memory management.

The Opal kernel abstractions are designed to support user software structured as groups of threads in overlapping protection domains, communicating through shared virtual storage and protected procedure call. Segments are the grain of virtual storage allocation, reclamation, and access control. All segments exist within a single virtual address space; virtual address pointers and resource capabilities are freely shared. Resource reclamation is handled with explicit reference counts, backed by bulk delete using resource groups, and enforced by accounting.

The Opal system interface is based on three fundamental principles.

- *No constraints on the language model.* Uniform referencing is supported without imposing any particular data model on users of the system: data entities are untyped virtual memory segments whose meaning is assigned at the language level. Portals and protection domains have no language semantics; they are minimal primitives on which language-level facilities such as protected objects and RPC can be built.
- *Efficient abstractions.* All abstractions are matched to what the hardware can efficiently support. Storage allocation, protection, and reclamation are coarse-grained at the operating system level. Fine-grained control is provided for language level objects by compilers and runtime systems, clustered above coarse system-level abstractions.
- *Orthogonal protection.* Memory protection in Opal (using protection domains) has been separated from other issues that are combined with protection (based on processes) in conventional systems. Protection is decoupled from: (1) program execution, through use of RPC as the basic mechanism for animating passive protection domains; (2) resource naming, through the use of context-independent capabilities based on portals; (3) resource ownership, through the use of resource groups; and (4) virtual storage, through the use of named segments in a global address space.

The intent of these choices is to build an efficient system general enough to accommodate a range of programming models, in which memory protection is cheap, easy to use, and easy to change. A user application is structured as a collection of threads, procedures, and data structures in the global virtual memory. Any thread can call any procedure or address any data, assuming it has adequate privilege to complete its actions. This permits an application to use a variety of *protection structures* while executing the same code over the same data. A protection structure is determined by how data is partitioned across segments, how protection domains are imposed over threads, which domains have access to which segments, and how resource allocations are assigned to resource groups. The protection structure determines how much it will cost to run the computation, and what types of unexpected behavior will be trapped or recovered from at runtime. The next chapter explains runtime facilities to simplify protection structuring and sharing for application programmers.

Chapter 5

Object-Based Sharing and Protection in Opal

The previous chapter described the basic Opal abstractions to support user software structured as teams of threads executing in multiple protection domains, interacting through shared memory segments in a common virtual address space. This chapter describes a runtime system with a simpler programming interface to these facilities, based on the object programming model outlined in Section 3.2.3.

An *object* is a typed data record named by a unique identifier and accessed through a procedural interface (Section 2.2.3). Objects are fundamental to many systems with sharing goals similar to Opal's; most of these systems have supported object naming, sharing, and protection directly in the hardware or operating system. In contrast, our approach employs object-based runtime support to allow seamless use of raw operating system facilities that are not themselves object-oriented: protection domains, cross-domain control transfers through portals, and access to shared and long-term storage through segments. The global address space is the basis for uniform naming and sharing of application objects, and the procedural abstraction of objects hides the details of managing protection and sharing within the global address space.

Opal's object-based runtime system allows programmers to organize sharing and protection with minimal impact on the logical structure and syntax of object-oriented programs. The primary goal is to support a range of alternative protection structures within a single application system, but without compromising the generality of code that is not directly responsible for configuring the structure. The object package supports programmers in four inter-related areas in which they must express protection structuring choices:

- **Memory Partitioning.** User code must partition procedures and data structures across segments. However, procedures that create data structures may be isolated from the details of data placement, so that they can be used generally.

- **Synchronization.** User code must synchronize access to data structures used by multiple threads, but synchronization facilities transparently handle mutual exclusion of threads in different protection domains.
- **Code binding.** Code to operate on shared data must be available to the domain of any thread with access to the data. The system can automatically bind domains to shared procedures for the objects accessed from that domain. Shared memory used in this way is *modular*: threads use the same code to operate on a given piece of shared data, to ensure common usage conventions for that data.
- **Managing protection boundaries.** User code must program control transfers across protection domain boundaries, to access protected data or to restrict untrusted code. Protection boundaries should appear as procedural interfaces rather than explicit portal switches.

The object facilities presented in this chapter consist of types, primitives, and conventions for programmers using the C++ language [Str86]. Our intent is to illustrate (1) the value of objects to simplify application use of sharing and protection in a single address space operating system, and (2) the value of a common virtual address space as a foundation for building efficient object abstractions outside of the operating system kernel. Many extensions for convenience and generality suggest themselves.

For our purposes, an object is a language-defined entity, occupying a contiguous sequence of bytes allocated from a heap. Its size and internal structure are statically determined and known only by its type implementation (*class*), which exists at runtime as a package of procedures, or *methods*. Objects are passive; they execute only when threads invoke their methods. (Of course, their methods may make them "active" by creating internal threads.) Objects may contain pointers (links or references) to other objects to form complex data structures. Accepted definitions of "object-oriented" often involve other features as well, for example, type system rules to permit method reuse among related types, or compatibility of types with similar interfaces. These issues are not essential to the present discussion; the ideas in this chapter can extend to any language with user-defined abstract types whose instances are heap-allocated and named by reference. In particular, our use of C++ is a marriage of convenience.

The Opal/C++ runtime system has three basic features:

- **Shared objects** are accessed directly in memory by threads in multiple protection domains. A client of a shared object names it with a virtual address and invokes its methods with ordinary procedure calls. Shared objects could be persistent or distributed objects in an extended global address space (see Chapter 8), imported into memory on the node of the invoking thread. Opal/C++ supports object sharing with facilities to partition objects across segments, synchronize across domain boundaries, and import objects and their methods automatically.

- **Protected objects** are also used by threads in multiple protection domains, but their procedural interfaces are enforced by hard memory protection boundaries. Protected objects are named with *password capabilities* that support automatic binding and object-grained access control. Their methods are invoked by protected procedure calls through portals.
- **Uniform access** to shared and protected objects. Creation and invocation of shared and protected objects are syntactically identical. There is no fundamental distinction between shared objects and protected objects; an object is ‘‘protected’’ only from the perspective of threads that are insufficiently privileged to attach the object and access it directly. The two styles of object access are made to appear uniform by means of *proxy objects* that hide the details of managing capabilities and protected calls.

This chapter outlines these object facilities in sufficient detail to understand their purpose, their use, and their relationship to the Opal kernel abstractions presented in the previous chapter. Section 5.1 deals with shared objects, which support modular, transparent access to shared and persistent memory. Section 5.2 deals with protected objects, which are used to introduce protection boundaries to enforce modular interfaces at runtime. Section 5.3 illustrates use of application use of shared and protected objects with some simple examples.

5.1 Shared Objects

A primary goal of many object-oriented systems is to allow applications to share or exchange complex linked data structures, or store them on disk and retrieve them incrementally as pointers are followed. These systems include the persistent object systems and object-oriented databases (Section 8.1) created primarily to support complex design applications (Section 2.1). These systems transparently import shared objects into active program contexts, primarily by (1) trapping references to nonresident objects, (2) copying those objects into the private virtual memory of the referencing process, and (3) translating pointers to implement uniform inter-object referencing in data used by multiple programs.

In Opal, support for object sharing (and, by extension, object persistence) derives directly from segment sharing in the global virtual address space. Objects are contained within segments: the address of a segment (and therefore the address of each object within it) does not change throughout its lifetime. Objects are shared by sharing segments; object access control is by segment access control. Shared objects are named by global virtual addresses; nonlocal references are trapped by MMU hardware and satisfied by ordinary page faults. The system can find an object anywhere in the system from its virtual address.

Opal's object runtime system and its associated thread package (OThreads [FCL93]) define a set of standard object types that can be used to organize and control object sharing. These types include primitives to control object clustering within segments, synchronize shared object accesses by threads executing in different protection domains, and import shared objects and their methods dynamically as pointers are followed. The following subsections discuss each of these facilities in turn.

5.1.1 Memory Pools

A *memory pool* is a self-contained heap for object allocations, occupying a contiguous range of virtual addresses within some segment. Each object is contained within a memory pool; linked object data structures may span memory pools. To allocate or free an object, a thread must be executing in a protection domain that has write access to the object's memory pool, i.e., the segment containing the pool is write-attached.

Memory pools are represented by the *MemoryPool* type (Table 5-1) with procedures to allocate and free heap slots. C++ applications do not invoke memory pools directly; instead they use the standard C++ *new* and *delete* operators, which have been redefined to work with multiple memory pools. The *new* operator creates an object by allocating a heap slot from the calling thread's *current memory pool*. A thread's current pool can be changed in a stack-like fashion using *PushMemoryPool* and *PopMemoryPool*. In this way, the thread can switch its memory pool for the duration of a procedure call; the invoked procedure can create objects in the standard fashion, without concern for where those objects are allocated. Objects are freed using the *delete* operator; if a garbage collector were present, it would make these *delete* calls implicitly. The *delete* operator determines the memory pool of the deleted object, and updates the pool to release the heap slot for recycling.

Most applications request heap memory with *NewMemoryPool*, which allocates a standard-sized segment and initializes it as a memory pool. Memory pools could also be used to control clustering within a segment to improve paging performance. That is, the application might create multiple small pools for subranges of addresses within a single segment, in order to specify that groups of objects likely to be accessed together should be allocated "nearby".

Memory pools have two additional features that are important but are left out of Table 5-1 for brevity. First, a pool can be placed in *packed mode* so objects are allocated in a log-structured fashion from adjacent areas of contiguous memory. Packed mode is fast and it eliminates internal fragmentation, but heap slots cannot be recycled until adjacent slots are released; it is intended for groups of objects that are allocated, traversed, and destroyed as a unit. Second, each pool has a *root*, an arbitrary pointer value set by user code to point to some interesting object in the pool, from which other objects may be reached. User code may assign symbolic names to objects in a pool by placing a directory at its root.

Table 5-1: Memory Pool Interface.

NewMemoryPool (out **SegmentRef**) returns **MemoryPool**. *Allocate a segment and initialize it as a memory pool.*

Thread.PushMemoryPool (**MemoryPool** pool). *Set a thread's current memory pool: all objects created by a thread are allocated from its current pool.*

Thread.PopMemoryPool (). *Restore a thread's current memory pool to its value at the time of the last PushMemoryPool.*

5.1.2 Synchronization

Operations within an object must be synchronized if multiple threads concurrently invoke its methods. Intra-object concurrency is inherent in our passive object model. OThreads supports standard facilities for synchronizing threads, including spin locks, blocking locks (mutexes), and condition variables. We assume that applications use these primitives to ensure that all multithreaded object methods are thread-safe. Of course, applications must do this whether or not these objects are invoked from multiple protection domains.

Shared memory complicates blocking synchronization for user-level threads. In the absence of sharing, a thread awakened by a blocking synchronization object can be made runnable with a short sequence of user-mode instructions. For example, the mutex release operation awakens a thread waiting for the mutex (if any) by unlinking the thread's descriptor from a list of blocked threads and linking it into its scheduler's list of runnable threads. Consider the following scenario for a shared mutex: thread **A** acquires the mutex from protection domain **A**; thread **B** blocks on the mutex from domain **B**; finally, thread **A** releases the mutex, but cannot access **B**'s scheduler to awaken thread **B**.

OThreads supports cross-domain synchronization with the *SharedMutex* type, which is interface-compatible with ordinary mutexes. Although a shared mutex is slightly less efficient for local blocking, it can be used to synchronize access to objects shared across domain boundaries. Link fields for the blocked thread list are allocated from the same pool as the shared mutex itself; they are accessible to any thread using the mutex. To awaken a blocked thread, *SharedMutex.Release* checks a field in the mutex blocked list to see if the thread uses the same scheduler as the releasing thread. If so, the blocked thread is awakened with a local call as before. However, if the blocked thread is foreign, it is awakened with a protected call to its scheduler in its domain. Each OThreads scheduler registers a portal for incoming requests to wake up threads in its domain. Access to threads and schedulers is controlled by key-based access checks, similar to the password capabilities described in Section 5.2.3.

5.1.3 Importing Objects and Methods

Like most object-oriented languages, C++ permits applications to define behaviorally abstract objects that bind to their methods at runtime: methods declared as *virtual* are invoked through a vector of procedure pointers (a *virtual table)* referenced by a pointer stored in the object. With runtime binding, a given method call site can invoke multiple implementations of an interface, depending on the implementation type of the target object. For example, new versions of a type can be introduced without affecting existing stored instances of old versions of the type (e.g., shared or persistent objects that outlast the life of a particular application).

Dynamic binding to virtual methods is central to fully transparent sharing and protection of objects in Opal/C++. In a single virtual address space, virtual methods for an object encountered in shared storage can be automatically imported. This ensures that all threads operating on a given object use the same methods, and therefore have a common understanding of the object's internal structure and usage conventions. Regrettably, we have not uses virtual methods in our current Opal prototype, because their implementation in previous versions of the Gnu C++ compiler assumed a private address space for the generated code. (The problem is instructive, and it is described in Section 7.3.2.) Still, it is useful to consider the intended role of virtual methods.

In Opal, the runtime representation of a C++ class is a set of procedures and virtual tables residing in a code segment called a *module* (see Section 7.1), which is typically accessible to any domain that trusts it to execute. The methods and virtual tables of classes linked for Opal exist at fixed addresses within the module, statically assigned when the module is constructed. Thus a sharable copy of the module can be located from the virtual table address in the object, using the *AttachByAddress* primitive in Section 4.6. This address can be used by a runtime fault handler to attach and load the module dynamically when a virtual method is invoked.

5.1.4 Summary

Opal's single virtual address space supports uniform referencing for shared objects and their methods. Any thread can invoke a shared object, given a pointer, knowledge of the object's interface, and authorization for the segment that contains it. With stacked memory pools, most procedures can create objects without concern for their placement in memory segments. With inter-domain synchronization (*SharedMutex*), thread-safe types can be written without concern for how instances of the type are shared. With dynamically bound methods, a component can use objects created by a different component and passed in shared storage. In short, shared virtual storage can be made transparent to subprograms built from properly synchronized, modular types. Sharing concerns can be confined to procedures that are directly responsible for configuring a protection structure in terms of segments and protection domains.

5.2 Protected Objects

This section describes runtime support for *protected objects* whose encapsulation is enforced by memory protection boundaries. A protected object is generally not directly addressable by its callers (*clients*); methods execute in a different protection domain (the object's *server)* with access to the object's internal data, invoked across the boundary using protected RPC through a portal. Object integrity is protected because the code that operates on the object's data is chosen by the object's implementor rather than by its clients. Protection works in the other direction as well: callers of a protected object are isolated from its methods. For example, a caller of an untrusted method can protect itself by invoking the object as a protected object in a restricted domain, as described in Section 5.2.5. Protected objects are the basic mechanism for programming uses of protection domains and portals in Opal/C++ programs.

The protected object facility supports capability-based naming and access control for protected objects. Opal's object runtime system implements capabilities with no specific support from the operating system or hardware, other than portals and the single virtual address space. (We also assume additional user-mode support for protected RPC through portals, described in Section 6.2.2.) In fact, these runtime-level capabilities are used for the Opal operating system interface itself: protection domains, segments, and resource groups are protected objects built using the facilities outlined in this section.

A primary goal of the protected object package is to make protection boundaries "translucent" to object-oriented programs. Programmers can reconfigure protection boundaries with minimal source code modifications, in order to achieve some desired balance between protection and performance. However, we shall see that some restrictions apply, for example, certain aspects of protection structure are determined at link time rather than at runtime in our current prototype. In any case, Opal's support for protected objects allows the programmer to inject protection boundaries in front of object-oriented interfaces, while preserving the application's uniform view of objects, and without affecting the modular structure of the program.

The following subsections describe the runtime support for protected objects, with an emphasis on how the nonuniform protection primitives are made to appear uniform to object-oriented user code, and how capabilities are minted, bound, transmitted, and validated at runtime.

5.2.1 Proxies and Guards

Although protected objects are named by capabilities, user code does not manipulate capabilities directly. A client invokes a protected object p through a *proxy* that contains the capability and masquerades as p. The proxy exports the same method interface as p, but its methods are stubs that make RPC calls across the protection boundary, passing the capability and method index as arguments. Similarly, a *guard* object for p handles the details of minting and validating capabilities on the server side of the boundary. The guard's methods are stubs that validate the incoming capability and arguments, and make ordinary procedure calls to the methods of p. The cost of this indirection is a moderate increment on the cost of a local RPC call across the protection boundary (see Section 6.3).

Proxies and guards effectively hide protection boundaries from both the client and the server. Ordinarily, neither the client nor the server is aware that a call originated or executed in a different protection domain. Proxies were introduced in [Sha86] for structuring distributed systems, and have appeared in many distributed systems, including Amber [CAL+89] and *network objects* [BNOW93] for Modula-3. Lipto, Spring and other systems have used proxies to hide protection boundaries and support configurable protection structures.

In general, a protected object may be an instance of any user-defined type. A protected object type is referred to as a *resource type*. Proxy and guard objects are instances of special types with method signatures matched to a specific resource type. The proxy and guard types are assumed to be generated automatically from the resource type definition by a trusted stub compiler. Proxy stubs in particular must be trusted, since they execute directly in the caller's context. Opal's proxy and guard types are currently "generated" by hand, but the protected object facility was designed to permit this process to be mechanized, as it is in other systems that use proxies. Note, however, that Opal proxies may pass hidden arguments to the server or cache immutable object state other than the capability. These extensions were designed to be compatible with automatic proxy generation, but they make the task more challenging.

5.2.2 Managing Proxies

Once a proxy type has been generated, proxy objects are created on demand by the runtime system as capabilities are imported as arguments and results of RPC calls. Proxy creation always occurs either in a stub method of some guard (if the capability is received as an argument to a call) or in the stub method of another proxy (if the capability was received as a result from a call). Three implementation details clarify the mechanics of handling capability arguments and results using proxies, and illustrate the tradeoffs involved:

- The stub must know the interface type of each capability imported at runtime in order to create a proxy for it. The expected type of each capability is determined at stub generation time from the static call signatures of the resource type methods. A stub creates the proxy for an incoming capability by calling a *resource type procedure* for the capability's expected type, addressed by a global symbol derived from the interface type name. One resource type procedure is generated for every resource type and statically linked into proxies and guards that import capabilities of that type. In a single address space, the address of the resource type procedure can be used as a type code, but there is currently no runtime type checking.
- The runtime system in each client seeks to create only one proxy for each capability, even if the same capability is imported multiple times. Before creating a proxy, the runtime system checks a table of known proxies hashed by capability. Note that a client may still have duplicate proxies, because they may be "discovered" in shared or persistent memory rather than created directly by that client. (Thus clients cannot compare pointers to test object equality.)
- Clients may pass their capabilities to other clients through RPC, by passing a proxy pointer as an argument to a protected call (i.e., a local call to some other proxy) or as a result from a protected call (i.e., as a return value to a call from some guard). The stub extracts the capability from the passed proxy, and transmits the capability instead of the proxy address. Currently, a proxy or guard stub faced with a proxy pointer as an argument or result will always choose to pass the proxy's capability rather than the proxy pointer itself. Thus there is no type-safe way to pass a raw proxy pointer; the stub will always convert the virtual address into a capability.

I am glossing over several delicate problems for fully uniform handling of proxies in C++. For example, since proxy creation is hidden from user code, it is difficult for a client to determine when to reclaim proxies. Also, sharing introduces new implementation concerns for proxies. In Opal, each proxy (and the capability within it) can be shared directly by sharing its containing segment, or it can be protected by allocating it from a private segment. This places some burden on the user code to set the current memory pool correctly when proxies may be created. These and other pitfalls can be handled, but in practice they compromise transparency somewhat; internal interfaces used as "fault lines" for injecting protection boundaries must meet certain unanticipated requirements. Some of these issues are discussed in detail in [Sch94].

Note that we have assumed that all protected calls that pass or return capabilities involve a stub of some other protected object. This has two implications. First, a would-be client must get an initial proxy by some means (e.g., through shared memory or from a name server at a statically known portal). Second, client requests to create new protected objects within the server are problematic, because C++ has no runtime class objects; instances of a class are created by *constructor functions* that are associated with the class but are not themselves methods of any object. By convention, a client creates a protected object by invoking the methods of some other protected object that executes the constructor within the server's context.

5.2.3 Capabilities and Guards

Opal/C++ proxies name their target objects with *password capabilities* similar to those proposed in [APW86] and used in several extensible operating systems (e.g., Chorus and Amoeba), as well as popular RPC services including Sun Microsystem's Network File System (NFS) [SGK+85]. As depicted in Figure 5-1, an Opal capability is a triple of 64-bit values: a portalID to identify the server domain and entry point, the global virtual address of a guard object within the server, and an access control key. A capability contains all of the information needed for the runtime system to transparently bind to the server and activate the object with protected, access-checked RPC calls.

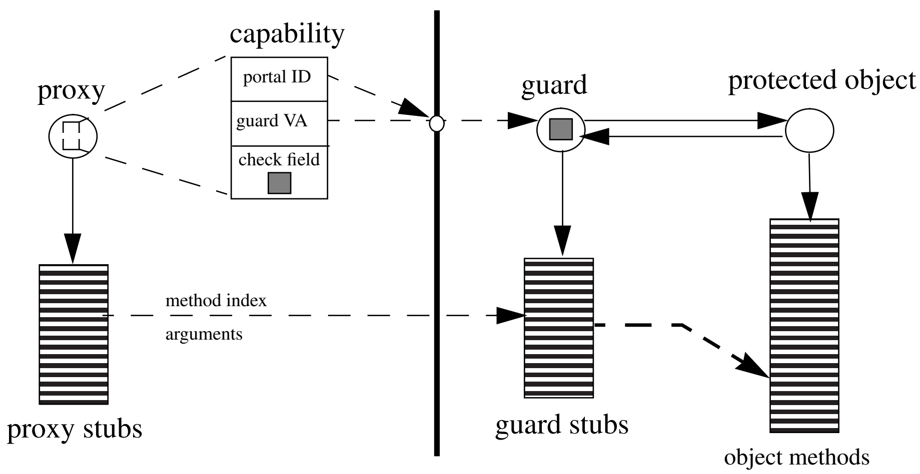


Figure 5-1: A protected object with a proxy and guard.

Guard objects are central to minting and validating capabilities. In particular, guard objects contain the access control keys or "passwords" used for authorization. The runtime system constructs a guard for a protected object *p* the first time user code passes a pointer to *p* as an argument or result of a protected call. (Again, this is always initiated by a stub method in some other proxy or guard.) The system initializes a *check field* in the newly created guard to a "random" number, which serves as the access key in all exported capabilities for *p*. Incoming capabilities are validated by the portal entry instruction sequence, which compares the access key with the check field of the guard addressed by the capability. Password capabilities cannot be forged because the value of the check field is probabilistically impossible for a client to guess, and there is no way to determine if a guess is correct except to try it, alerting the server to the attempted forgery.

Figure 5-1 illustrates the structure of a protected object and its guard. Every resource type is a subtype of a system class called *ResourceObject*, which adds several fields to each protected object, including the address of its resource type procedure (to create a guard), and the address of a guard if one has already been created. Each guard contains a pointer to a method vector for its stubs; the portal entry sequence branches to a guard stub by probing this vector with a method index passed in each RPC call.

Capabilities permit dynamic binding to RPC interfaces. In most RPC systems, clients bind to a service explicitly by presenting a symbolic name for the service to a *binding clerk*. In Opal, a client binds to an anonymous server implicitly on the first call to any of the server's objects, using the portalID embedded in the capability. By default, the runtime system registers one portal for each protection domain, and includes its portalID in all capabilities exported from that domain. The runtime system registers an additional portal for each object designated by user code as a *ServiceInstance*; that portal serves any calls on the service instance, and on all protected objects created by those calls. This can be used to combine logically independent services in the same protection domain -- or separate them into different domains -- without invalidating existing capabilities.

I emphasize that application programmers are not concerned with these details of proxies, guards, and capabilities. Opal uses capabilities to support flexible, dynamic, access-controlled RPC relationships among protected components. Opal uses proxies in order to free programmers from explicitly managing capabilities or protection boundary crossings. The examples in Section 5.3 should clarify this point.

5.2.4 Discussion: Password Capabilities

Password capabilities offer probabilistic rather than absolute protection. However, in some respects they are "safer" than traditional capabilities. While additional access checks are generally unnecessary, they are of course not prohibited. A server could choose not to grant access even if a client presents a valid capability, perhaps basing this choice on an authenticated token provided by the RPC layer. Note also that password capabilities can be revoked by changing the value of the check field in the guard.

Opal's use of runtime-level password capabilities contrasts with many systems, ranging from Hydra to Mach to Microsoft NT, that include protected object support in the operating system kernel. In these systems, capabilities are maintained in a kernel-controlled object table for each protection domain.

Password capabilities have several important advantages for Opal. Most importantly, they are self-describing and context-independent; like virtual addresses, they can be passed and shared among protection domains without system intervention. Since capabilities are simply values in memory, a thread's protection domain determines the capabilities available to the thread by determining the memory it can access. Password capabilities can be placed in long-term storage as well, given additional system support for recoverable servers with stable portal assignments, as outlined in Section 8. In addition, password capabilities can offer better performance than kernel capabilities, which involve the kernel each time a protected object is created or destroyed, and each time a capability is added or removed from a domain. In Opal, minting capabilities involves a single kernel operation to create a portal, and this cost is amortized over any number of protected objects served through the portal.

One advantage of kernel capabilities is that they can be reference-counted automatically by the kernel. This does not solve the problem of reclaiming objects in cases where capabilities are shared by threads within a domain, as any Mach programmer can attest. Nevertheless, every Mach capability is eventually reclaimed when the domain that owns it is destroyed. In Opal, this need is filled by resource groups. We believe that resource groups are necessary for safe, general use of password capabilities, particularly in a system supporting persistence. (However, we have not extended Opal's resource group mechanism to encompass user-defined protected services.)

In short, password capabilities support sharing and persistence, and they promise superior performance for the majority of cases where probabilistic protection is adequate; in addition, they free the kernel from the need to support objects and capabilities.

5.2.5 Proxy Constructors

To this point, I have presented protected objects as if they were only used by clients and servers that have already been activated by other means. Protected objects can also be used to set up and activate protection domains. The basic idea is that a newly created domain (a *child*) is activated by installing an object in some segment, attaching the segment to the child, and invoking the object as a protected object so that its methods execute in the child. Runtime facilities to create and invoke protected objects form the basic mechanism for animating protection domains in Opal/C++.

Protected objects are a general and flexible alternative to the notion of "executing a program" common to private address space systems. In Unix-like systems, user code executes an untrusted procedure by creating a process with a private address space, loading it with a program image, and invoking a single static entry point (*main*) defined in the image; the process destroys itself when the main procedure returns. In contrast, Opal/C++ programs can impose protection domains over a connected object graph in a range of configurations. Programs create objects, invoke their methods one or more times, possibly pass their references to other objects, and eventually destroy them. If protection is desired, perhaps because an object's methods are not trusted, the object is treated as a protected object and invoked through a portal to a restricted domain. References to the child's objects may also be passed to other domains, to create sharing relationships and RPC connections among a group of protected application components. We defer an example to Section 5.3

To support this object-oriented approach to structuring protected interactions, the proxy generator emits one or more *proxy constructor* procedures for each resource type. Each proxy constructor is similar to a user-defined constructor for the resource type itself, but it takes a protection domain descriptor as an additional argument, and it returns a proxy for the newly created object, rather than the object itself. Calls to the proxy cause the object methods to execute within the specified protection domain.

Proxy constructors are used in conjunction with runtime system primitives for managing protection domains, presented in Table 5-2. *OpenDomain* creates a child domain with a guarded portal and a new memory pool that is attached to both the child and the parent. This involves four Opal system calls to create the child domain, register a portal, and create a segment and attach it to the child, but let us suppose that these are combined in a single compound operation. Once the child domain is "open", the proxy constructor executes a canned code sequence to: (1) invoke the ordinary constructor to create an object *p* in the shared pool, (2) create a guard for *p*, also in the shared pool, and (3) create a proxy for *p* in the caller's current pool. The constructor executes entirely at the runtime level in the parent context, over memory shared with the child. Note that this is safe for the parent because no threads are active in the child.

Table 5-2: Runtime interface for managing child protection domains.

OpenDomain () returns **Domain**. *Create a protection domain with one guarded portal, and a memory pool shared by the child and parent.*

ReapDomain (**Domain** child). *Destroy the child protection domain, ceasing execution of all threads, but leaving its descriptor (proxy) and shared memory pool intact.*

CloseDomain (**Domain** child). *Detach the child's shared memory pool from the parent, and destroy the child domain descriptor.*

5.2.6 Summary

This section outlined Opal's runtime system support for protected objects, which can be used to export a service to mutually distrustful clients, or to safely execute untrusted code. Global naming and access control for protected objects is provided by runtime-level password capabilities, based on Opal's uniform name spaces for portals and virtual addresses. Proxies and guards hide protection boundaries and capabilities from an object's implementation and from its caller, to combine diverse styles of object access in a single system that is both efficient and uniform. Our use of proxies and guards reflects our view that low-level interfaces should be designed for performance and generality, with language and runtime support to create a uniform programming interface matched to the programmer's language model.

5.3 Protection Structuring with Shared and Protected Objects

The protected and shared object facilities described in the previous two sections constitute a complete (if minimal) programming environment for Opal, with respect to communication, protection, and sharing. In particular, the notions of creating a process and executing a program are replaced with primitives to seamlessly impose protection domains over a graph of objects linked by pointers, interacting through procedural interfaces. Shared objects and protected objects can be used to structure rich RPC interactions and sharing relationships among these protection domains.

To illustrate how these facilities are used for protection structuring, and some of the issues involved, this section presents a simple protected object type called *Child* (Figure 5-2) and shows how it is used to construct several alternative protection structures. These idealized examples ignore certain details, such as how code modules are attached to protection domains. Let us suppose that all procedures are globally accessible and attached on access faults. Several other issues emerge as the examples develop.

```
class Child: ResourceObject {

    internal data

    SharedMutex mutex;
    int count;

    method signatures

    VisitSibling (Child* sibling);
    Visit();
}
```

```
Child::VisitSibling (Child* c) {
    c->Visit ();
}

"Visit" a Child object and advance its counter.

Child::Visit () {
    mutex.Lock();
    count = count + 1;
    mutex.Unlock();
}
```

Figure 5-2: Code for the *Child* protected object class.

Children are constructed by the *SpawnChildren* method of a *Parent* object. Figure 5-3 presents the code for the simplest version of *SpawnChildren*, but the other details of the *Parent* class are not important. When *SpawnChildren* is invoked on a *Parent*, it creates two children and passes each a reference to the other. The children then interact directly by invoking each other's *Visit* method. In the initial version of *SpawnChildren*, the children are created in the same segment, and all objects are invoked with ordinary procedure calls within a single protection domain. The examples develop three new versions of *SpawnChildren* with different protection structures. In all examples, *SpawnChildren* is responsible for setting up the protection structure, but the *Child* class is unchanged.

```
Parent::SpawnChildren () {
      Child* right, left;

      right = new Child ();
      left = new Child ();
      right->VisitSibling (left);
      left->VisitSibling (right);
}
```

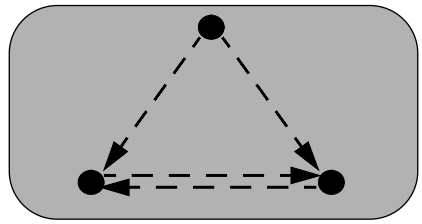
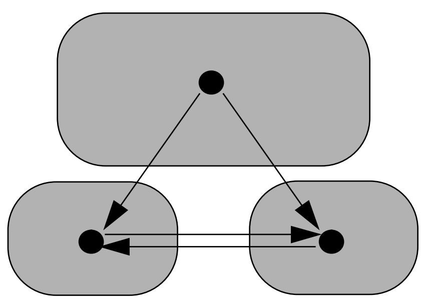


Figure 5-3: Code for the "basic" version of *SpawnChildren*.

Figure 5-4 shows the modified version of *SpawnChildren* using *OpenDomain* and proxy constructors to execute child methods in separate protection domains, with no access to each other's or the parent's data. All method calls in this example are protected calls through capabilities. Many useful variations of this simple structure are possible. For example, the parents and/or children may make repeated calls to one another, to sustain interactions over a longer period. The parent may grant the children read-only access to some of its data, or it may create segments to preserve the children's output data for further processing after the children are destroyed. In this structure the parent chooses the code that executes in the children, but the children cannot affect each other except by invoking interfaces determined by the parent. Neither child can affect the parent in any way unless the parent permits it.

```
Parent::SpawnChildren () {
      Child* right, left;
      Domain* d1, d2;

      d1 = OpenDomain ();
      d2 = OpenDomain ();

      right = new Child (d1);
      left = new Child (d2);
      right->VisitSibling (left);
      left->VisitSibling (right);

      CloseDomain (d1);
      CloseDomain (d2);
}
```

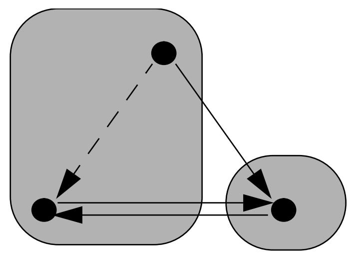


*This version of **SpawnChildren** creates two child objects in separate segments; the methods of each child execute in a private protection domain.*

Figure 5-4: Creating protected objects with *OpenDomain* and proxy constructors.

In Figure 5-5, we modify this structure to permit one child to upcall methods in the parent, for example, to notify it of a change in the state of shared data. The parent creates the left child as a local protected object, and passes its right child a capability for the left child. To achieve this structure in our Opal prototype, the two children must actually be of different types. This is because proxies are statically linked, thus a compiled call site for a protected object invocation statically branches to a different entry point than a call site for a shared object of the same type. I briefly digress to discuss this restriction in more detail.

```
Parent::SpawnChildren () {
    Child* right, left;
    Domain* d;

    d = OpenDomain ();
    right = new Child (d);
    left = new Child ();
    right->VisitSibling (left);
    left->VisitSibling (right);

    CloseDomain (d);
}
```



This version of SpawnChildren creates the right child with its own domain, and passes it a capability for the left child. The methods of the left child execute in the parent's domain.

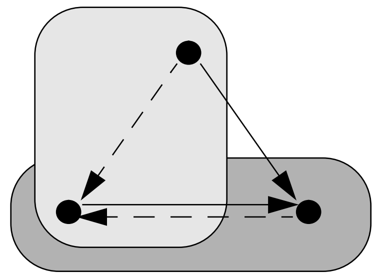
Figure 5-5: Upcalls from a child object in a restricted domain.

The same restriction exists for a different reason in Psyche [SLM90] and Mungi [HERV94], which implement configurable protection boundaries by driving transparent protected calls from traps caused by jumps to procedures that are not accessible to the calling domain. In these systems, procedures are partitioned among domains; procedures rather than objects are the protected entities. Proxies may be more efficient for the protected calls, since they do not incur a memory access trap. We originally chose to use proxies because they can be more flexible as well as more efficient than the trap-driven approach. In the intended design, proxies are compatible with their resource types, so that a proxy can be substituted for a local object and vice versa. Furthermore, all proxy methods are virtual functions, so that this substitution can occur at runtime. The executable code is independent of protection, as in the trap-driven approach, and the protection boundaries are configurable on a per-object basis at runtime. The C++ language features of inheritance and virtual functions are expressly designed to support this sort of arrangement, but we have not used them due to linking difficulties with our compiler's chosen implementation of these features (Section 7.3.1), hence the restriction of link-time selection.

Returning to the example, the protection structures explored so far both limit the interactions of the parent and its children to protected calls, which are expensive. In the alternative arrangement depicted in

Figure 5-6, the parent and its right child can interact by invoking methods of a shared object, in this case the *Visit* method of the left child.

```
Parent::SpawnChildren () {
      Child* right, left;
      Domain* d;
      MemoryPool *mp;
      SegmentRef *srp;

      d = OpenDomain ();
      right = new Child (d);

      mp = NewMemoryPool (&srp);
      PushMemoryPool (mp);
      left = new Child ();
      d->Attach (srp);

      right->VisitSibling (left);  /* RPC */
      left->VisitSibling (right);  /* local */

      CloseDomain (d);
}
```



This version of SpawnChildren creates the right child with a private domain; the left child is shared by the parent and the right child's domain. The methods of the left child execute in the caller's domain.

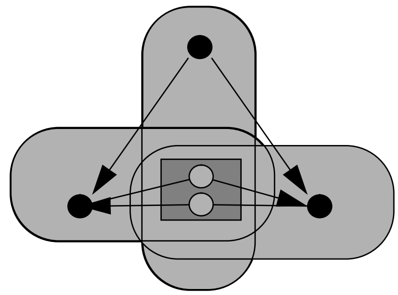
Figure 5-6: Interacting with a child domain through a shared object.

In this example, the data and methods of the left child are shared; calls to *Visit* execute within the caller's domain, but operate on shared data within the left child. If multiple threads were active in this example, concurrency in the shared child would be controlled by the shared mutex, as descibed in Section 5.1.2. Again, the child objects must be of different types in order to achieve this structure in our prototype, due to link-time selection of the *VisitSibling* stub.

The final version of *SpawnChildren* shown in Figure 5-6 illustrates passing a capability through shared memory by sharing a proxy. The parent switches to a new memory pool before creating the children; the proxies are then shared by attaching this pool to the right child's domain. The right child is unaware that it has been passed a proxy rather than the object itself. This example, like the others, ignores the restriction of link-time proxy selection. In addition, passing a proxy pointer as a virtual address is problematic, as explained in Section 5.2.2, since the default stubs will convert the pointer to a capability, causing a new proxy to be created in the right child. However, proxies are easily shared in our prototype by linking them into some other shared data structure.

Shared proxies involve some special implementation issues ignored in this treatment. The basic problem is that proxies contain domain-specific state needed by the RPC layer. Shared proxies were implemented by Rene Schmidt as part of his MS thesis work [Sch94], which extends the protected object facility to improve performance for read-mostly RPC services. The idea is that the service exports objects in mem-

```
Parent::SpawnChildren () {
        Child* right, left;
        Domain* d1, d2;
        MemoryPool *mp;
        SegmentRef *srp;

        mp = NewMemoryPool (&srp);
        PushMemoryPool (mp);

        d1 = OpenDomain ();
        right = new Child (d1);
        d1->Attach(srp);

        d2 = OpenDomain ();
        left = new Child (d2);
        d2->Attach(srp);

        right->VisitSibling (left);
        left->VisitSibling (right);

        CloseDomain (d1);
        CloseDomain (d2);
}
```



This version of SpawnChildren creates both children in separate domains, and passes each child a pointer to a shared proxy for the other.

Figure 5-7: Passing capabilities through shared memory by sharing proxies.

ory segments shared read-only with certain clients. Client queries are handled efficiently by executing server code within the client domain, accessing the objects directly in readonly memory. Client requests to modify the objects are handled by protected calls to the server domain, as before. Readers in the untrusted client domains synchronize with writers in the server domain using a form of optimistic version-based synchronization. This facility has been used to build an Opal-hosted name server in which operations to change the name space are protected calls, but the more common lookups complete in shared memory. As with the examples in this section, the difference is transparent to clients; the details are encapsulated within the shared proxies.

These examples are chosen to illustrate both the flexibility of shared and protected objects, and the difficulty of achieving fully uniform use of that flexibility in practice, when different objects of the same type are used and shared in different ways. The solution is not fully uniform for some programs, given the shortcomings we have exposed, primarily because the protection structuring is selected at link time. Nevertheless, these examples show that modular C++ programs can be configured to run with a range of protection strucures, as a collection of protection domains cooperating with RPC and shared memory, affecting only the source code directly responsible for configuring the structure (*SpawnChildren*). While the author and readers may suspect that this result is primarily of academic interest at present, the ability to reconfigure the protection structure serves as an important measure of transparency

5.4 Summary

The concept of abstract objects in a uniform object name space has long been used to promote sharing of data and services in computer systems. Because objects are *self-describing* (their methods are dynamically bound), applications can operate on objects encountered in shared or persistent storage. Because objects are *encapsulated*, they can represent protected services invoked across a network or protection boundary. Because objects are *abstract*, object method invocation is syntactically and semantically independent of the mode of access and the object's location in the system.

This chapter described runtime-level facilities for object-oriented sharing and protection in the Opal environment. This package shows that Opal's kernel facilities can be used naturally by object-oriented programs. From an application programmer's perspective the object runtime system serves three basic purposes:

- **Support for modular, transparent shared memory.** Objects can be allocated from memory pools in shared segments and synchronized with shared mutex objects. Shared objects are naturally extended to automatically import method code; this ensures that all threads accessing an object use the same code to operate on it, transparently enforcing usage conventions on the shared object. Shared objects add no overhead to ordinary shared memory, beyond the small price of object-oriented programming (i.e., more heap activity and more procedure calls).
- **Support for transparent use of memory protection**. Opal/C++ programmers can seamlessly insert protection boundaries along the ‘‘fault lines’’ of procedural interfaces in an application at compile time or link time. Cross-domain control transfers through portals are concealed behind object interfaces, with the small cost of indirection through proxy objects.
- **Uniform capability-based naming and access control.** The protected object facility provides a capability mechanism for user-defined protected services, and a uniform name space for all protected system resources. Password capabilities retain their meaning when they are stored in shared or persistent memory.

An important theme of this dissertation is that system support for object sharing, object persistence, and object protection should be confined to the language and runtime system, layered above generic kernel mechanisms for untyped storage and communication. Opal's operating system abstractions are designed to support efficient object-oriented sharing environments, as well as other programming environments that are not object-oriented. The runtime-level object support outlined in this chapter relies on Opal's kernel features, notably (1) pure protection domains, (2) globally named portals, and (3) uniformly sharable segments within a single virtual address space.

Chapter 6

An Opal Prototype

This chapter describes a prototype implementation of the Opal facilities discussed in the previous chapters. The goals of the prototype effort were (1) demonstrate that the Opal model can be implemented simply and efficiently, (2) concretely illustrate the issues and techniques involved, and (3) produce functioning software for further experimentation and development.

For expediency and portability, we built on existing software whenever possible, to narrow our focus to those aspects that are central to Opal itself. In particular, we chose to implement the Opal kernel as a server above a standard microkernel, the Mach system from CMU [JR86]. Like other microkernels, Mach offers general, low-level primitives intended to support diverse operating system environments or "personalities" (e.g., Unix or DOS) as application-level subsystems. This chapter shows that these primitives can also host an Opal environment in a straightforward way with acceptable performance above a standard microkernel. This approach was expedient because it allows our Opal prototype to coexist with the Unix implementation also hosted by Mach, and to use Unix-based tools to bootstrap and monitor the Opal environment, as well as to construct and debug Opal programs and the Opal system itself. However, given certain modifications to the microkernel, particularly in the lowest level of the virtual memory system (the *physical map*), we do not believe that a native Opal kernel implementation would offer compelling performance advantages over a microkernel-based implementation.

The Mach-based Opal prototype has three major pieces:

- The *Opal server* supports the basic kernel interface (segments, protection domains, portals, and resource groups) described in Chapter 4, and controls allocation and access for the global virtual memory.

- The *standard runtime package* provides low-level facilities for executing user code and communicating with the Opal server and the external environment through portals. It also supports the C++ language interface discussed in Chapter 5, including threads, shared mutexes, capability-based RPC, proxies, and heap management.
- A set of custom *linking utilities* statically link code modules to execute at permanently assigned addresses in the persistent virtual address space. Chapter 7 discusses Opal's approach to linking, and describes the prototype linking facilities.

The Opal prototype is implemented mostly in C++ with about 30,000 lines of code. We built a standalone system from March of 1992 through April of 1993, then extended it from January through June of 1994 to support the foundations of the persistent network virtual memory proposed in Chapter 8. The prototype was developed on 32-bit DECstation platforms based on MIPS R3000 processors, then ported to 64-bit DEC Alpha-based systems as they became available. A number of graduate and undergraduate students participated in this effort at various times (see the acknowledgements).

Section 6.1 and Section 6.2 discuss the internals of the Opal server and standard runtime library respectively. Section 6.3 presents some performance measurements from our prototype, and Section 6.4 describes an experiment with application structuring on our Opal prototype.

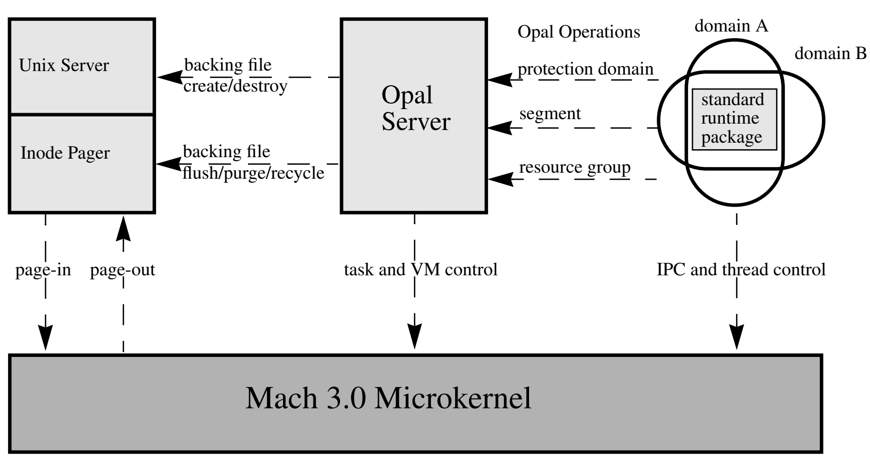


Figure 6-1: Organization of the standalone Mach-based Opal prototype.

6.1 The Opal Server

The Opal server runs above an unmodified Mach 3.0 microkernel. The Mach kernel interface includes primitives to create addressing contexts (*tasks*), manage their virtual memory mappings and access permissions, create threads within those contexts, and pass messages through *ports*. Mach systems also permit access to the network and disk through the Unix subsystem, and allow virtual memory-mapped access to user-defined storage resources (*memory objects*). Backing storage for Opal's global virtual memory is supplied by an independent file server; segment pages are paged to and from backing files using the Mach *external memory management* interface [YTR+87]. All Mach objects -- including kernel objects, memory objects, and ports -- are named by capabilities (*rights*), integers that index per-task object tables in the kernel. This section assumes some familiarity with these Mach facilities used by the Opal implementation.

The Opal server is a Mach task that interacts with the microkernel to create the Mach objects (tasks, ports, threads, and memory objects) used to implement the Opal abstractions. The server presents external RPC interfaces to the primary Opal kernel resources (protection domains, segments, and resource groups), which are protected objects named by password capabilities and invoked through guards as described in Chapter 5. Opal applications never use the Mach primitives directly; they make calls through the Opal runtime package to the Opal server, which holds rights for all related Mach objects in its internal data structures.

The next few subsections describe the key aspects of the Opal server: management of the single address space, backing storage and paging for Opal segments, and the implementation of protection domains, cross-domain communication, and resource groups.

6.1.1 Segments, Backing Files, and Segment Pools

A primary role of the Opal server is to partition the global virtual memory into segments, and manage the address space and backing storage for those segments. The server must allocate variable-size address ranges to create new segments, recycle released address ranges, look up segments by virtual address, and bind segments to backing files if their data is paged out.

The *SegmentPool* module manages a block of contiguous address space as an allocation pool for segments, using the interface in Table 6-1. In a distributed Opal system with a network-wide address space, different nodes allocate segments from multiple, disjoint, fixed-size pools backed by a shared file server and indexed by a global table, as described in Section 8. In a standalone system there is a single pool containing all available address space.

Table 6-1: *SegmentPool* internal interface.

NewSegmentPool (**Address** start, **ByteCount** len, **FilePool** back) returns **SegmentPool**. *Create a **SegmentPool** for a given address range and backing file pool.*

SegmentPool.CreateSegment (**ByteCount** count, **Boolean** aligned) returns **Segment**. *Allocate a segment object and backing file, and associate it with a free address range of the requested size and alignment.*

SegmentPool.Resolve (**Address** virtual) returns **Segment**. *Look up a segment by any contained address, by probing a splay tree of segments allocated from this pool.*

SegmentPool.DestroySegment (**Segment** seg). *Destroy a segment that has no remaining attachments or **SegmentRef** objects.*

Figure 6-2 illustrates the data structures that subdivide and manage address space within each *SegmentPool*. New address ranges for segments are carved from a table of free ranges sorted by address, using a first-fit policy. When a segment is released, its range is placed on an unsorted idle list, which is periodically merged into the free table; in this way, segment release is efficient and address recycling is delayed to minimize dangling references. Allocated segments are represented by *Segment* objects linked into a self-balancing binary splay tree [ST85] keyed by start address. These range trees are derived from the Mach kernel structures that index the capabilities held by each task. Splay trees accommodate variable-sized segments, and they support fast lookups: for a pool with *n* allocated segments, *Resolve* retrieves the segment containing a virtual address with *log(n)* comparisons in the worst case. In our prototype the splay tree is used primarily for client calls to *AttachByAddress* for published segments.

Segment deletion is controlled by reference objects with protected reference counts (Chapter 4.5.2). Each *Segment* points to a nonempty list of associated *SegmentRef* objects, each with a separate guard and independent reference counts manipulated by clients using the *Reference* and *Release* primitives. New *SegmentRef* objects are created by *Clone* requests from clients, and destroyed when their reference counts transition to zero. A segment is released when its last *SegmentRef* is destroyed. Each *SegmentRef* has a separate reference count to record attachments; a segment can never be released while it is attached to a domain.

Each *Segment* points to a *BackingFile* object representing the disk file that backs the segment. Backing files are one-to-one with segments. Each *SegmentPool* has one *FilePool* that manages a directory in a file system as a repository for backing files. *SegmentPool* calls the *FilePool Allocate* primitive to create empty backing files within the directory as needed. Each file's name is synthesized from a unique (within the pool) integer FileID assigned when the file is created; this is important for persistence, as described in Section 6.1.6. There are several versions of *BackingFile* and *FilePool* using different file services, as explained below.

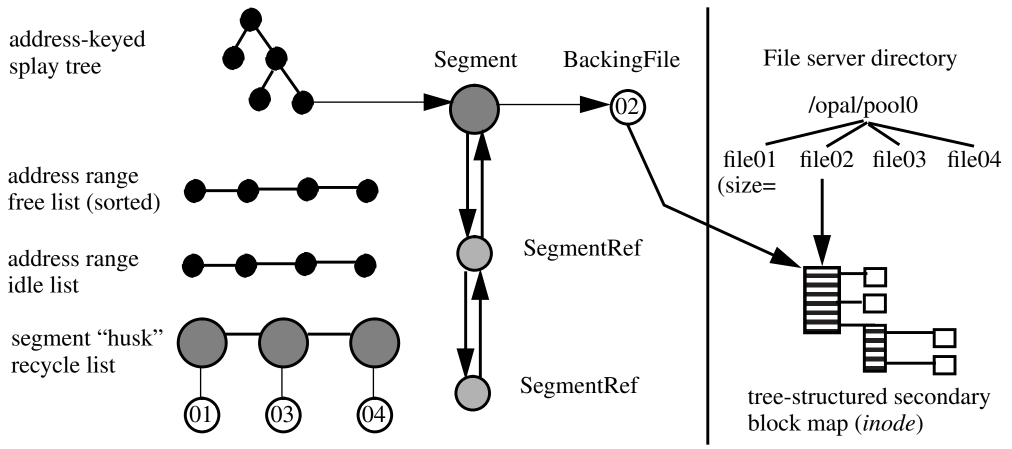


Figure 6-2: Internal structures and mapping tables for *SegmentPool* and *FilePool*.

When a segment is released, *SegmentPool* links its segment husk and backing file into a scrap list. If any of the segment’s pages have been written to the backing file, the file is returned to its initial state by truncating it. New segment objects and their backing files are recycled from the scrap list to speed both segment allocation and segment deletion. In particular, disk files are rarely created and destroyed in steady state; the Opal server normally calls the file server only to handle paging activity (e.g., because physical memory is inadequate to meet demands). Given adequate physical memory and an established pool of backing files, segment creates and destroys complete with a few splay tree and list operations in memory.

6.1.2 Backing Storage

The backing files for Opal segments are mapped into virtual memory and paged by an external paging server (*pager)* using the Mach external memory management interface. When a segment is first attached, the pager allocates a Mach port; the Opal server caches a right for this port in the *BackingFile* object and passes the port right as a memory object in a *vm_map* call to the Mach kernel for each segment attach. The kernel handles page faults on the segment by sending messages to this memory object port; the pager responds with a message containing the desired page by reference. Similarly, if the kernel opts to discard one of the segment’s pages from primary memory, it sends a message that prompts the pager to write the page to the segment’s backing file and deallocate it. In general, the pager satisfies the initial page fault on each page by zero-filling a page of memory, with no effect on the backing file; each backing file is written only to push active pages out of memory, and read only to retrieve those pages when they are referenced again.

In the configuration used for most of our experiments, memory object ports for Opal segments are created and served by a modified version of the standard Mach paging server, the *inode pager.* We modified the inode pager to return memory object ports for Unix files to the Opal server, so that it can attach Opal backing files directly to the tasks under its control, bypassing the Unix *mmap* operation. This has several benefits. First, the modified inode pager can short-circuit some overhead for Opal backing files because they are never accessed through ordinary Unix system calls while in use by Opal. Also, Opal backing files can grow physically through pageouts, and they are recycled with a new inode pager operation that (1) invalidates any of the segment's pages in memory, and (2) truncates the backing file. The backing file and memory object can then be reused cheaply for a different segment.

In a more recent configuration, segments are backed by a central file server accessed through Sun Microsystem's Network File System (NFS) protocol [SGK+85]. Opal does not use an NFS facility provided by Mach; rather, the Opal server itself implements a subset of the client-side NFS protocol, using Unix UDP sockets to communicate with the NFS server through the network. The NFS server supplies unique tokens (*file handles*) that name NFS files and directories; one token is cached in each *BackingFile* and *FilePool* object. NFS backing files are allocated and destroyed with NFS *create* and *unlink* operations on the file pool token, and recycled with NFS *truncate* operations. The Opal server delegates pagein and pageout requests on a segment by issuing *read* and *write* NFS calls on the file handle in the segment's *BackingFile* object. As an artifact of the current Mach networking structure, each page is copied up to four times; updated networking software could move data directly between the physical page frames and an AN2 network interface on some Alpha platforms.

One problem with NFS paging is that the pager does not know which pages within a file are empty (holes). Suppose the Opal server writes page x, then receives a fetch request for page $y < x$. If y has never been referenced, then the cheapest response is to zero-fill a local page. But the server does not know if y already contains valid data, since y is within range of the file (i.e., y is less than the file's size). It forwards the request to the NFS server; if y is a hole (no valid data), then the NFS server returns a zero page over the network. This cost is rarely incurred because pages beyond the end-of-file are known to be holes, and our heap manager fills segments sequentially. However, some heap slots (e.g., thread stacks and global data blocks) contain multiple pages filled over a period of time, and may have hidden holes.

6.1.3 Protection Domains

The Opal server implements protection domains directly as Mach tasks. Each domain is represented by a *Domain* record containing rights to the associated task, a list of registered portals (its *portal set*), and a list of *SegmentRef* objects for its attached segments (its *attach list*). *Attach* and *detach* operations update the attach list, and issue Mach *vm_map* or *vm_deallocate* calls to map or unmap the segment's memory object from the domain's task at the segment's assigned addresses. All MMU management for attached segments is handled by the Mach kernel, which moves the segment's pages in and out of memory by upcalling the paging server for the segment's memory object.

All cross-domain calls, including calls to the Opal server, are coded as control transfers through portals. However, we have not modified the Mach kernel to support native portals. Instead, the Opal runtime system simulates portal switches using Mach message primitives. The Opal server creates one Mach port (the *domain port*) for every protection domain; all portal switches into the domain are simulated by sending messages to this single port.

In our prototype, every protection domain is created with a separate instance of the standard runtime package, in an *initial data segment* allocated with the domain. The Opal server initializes protection domains in several steps.

- Attach the domain's initial data segment to the Opal server task, and preload it with standard runtime system data structures, including a user-mode thread scheduler and structures to manage RPC connections to other domains, as described in Section 6.2.2.
- Attach the standard runtime code segment to the new domain, and load its global static data (Section 7.3.1) into the initial data segment.
- Prime the thread package and IPC system by creating the domain port and one initial Mach thread blocked to receive messages on the domain port.

Since the Opal server creates all protection domains in this same initial state, it preallocates domains and caches them to reduce the latency of domain creation requests. However, domains are not recycled in the same way that segments are, rather they are simply destroyed due to the expense of restoring them to their initial state.

6.1.4 Portal Service

Naming and resolving of uniformly numbered portals is handled by the *portal service* module of the Opal server. Portal names are allocated from the virtual address space; the portal service allocates blocks of unbacked and unattached address space from *SegmentPool*, and parcels the individual virtual addresses out as portal IDs. These addresses do not point to any data; dereferencing a portal ID would generate an unresolvable page fault. However, portal IDs allocated from the address space are guaranteed to be unique, even in a distributed address space system.

No Mach kernel action is needed to create a portal. To register a new portal for a domain, the portal service allocates the next free portal ID and links a *Portal* record into the domain's portal set. All *Portal* records are also linked into a central hash table. When a user thread first requests to switch through the portal, the thread's runtime system calls a *ResolvePortal* operation in the Opal server, passing the requested portal ID. The server hashes into its portal table and returns send rights to the domain port of the portal's domain. The runtime system caches the portal-to-port association to short-circuit subsequent lookups, as described in Section 6.2.2.

6.1.5 Resource Groups

Like portals, resource groups are internal to the Opal server and have no corresponding Mach resources. Resource groups are used only to organize other resource records within the Opal server, so that clients may request deletion of related resources. Each resource group is logically an account to which resources are charged, represented as a *ResourceGroup* object with lists of resources charged to the group. At present, each *ResourceGroup* has two resource lists, one for protection domains and another for segment references (*SegmentRefs*). Every *Domain* and *SegmentRef* is charged to exactly one resource group, and holds a back pointer to its group.

ResourceGroup records are organized as a collection of trees with unbounded branching factors, representing subgroup hierarchies. A new subgroup is created by allocating a *ResourceGroup* and linking it into a list of children of its parent group. Deleting a resource group causes all resources in its lists to be released. If the deleted group has subgroups, they are released as well, along with their resources and descendent groups.

6.1.6 Persistence

The Opal server is designed to support persistence. That is, Opal resources including segments, protection domains, and resource groups, can survive across system restarts; persistent segments continue to occupy their previously assigned regions of the address space, and capabilities for Opal resources continue to be valid. Chapter 8 discusses some of the many issues raised by support for persistence, but we outline basic aspects of the implementation here. The key idea is that the Opal server's internal data structures, like all data in Opal, exist in segments whose backing files are retained in the file server across restarts. This section explains how internal Opal state and Mach kernel state are re-established after a restart, *assuming* that updates to Opal server structures are propagated to the backing files in such a way that the data can be recovered in a consistent state. In the general case this requires that some of the server's segments be transactional, a focus of continuing research; the current Opal prototype supports persistence only for planned shutdowns, not for failures.

The Opal server's internal data structures are rooted in a designated *root segment*. On a restart, the root segment is recovered from its backing file and reattached at its assigned address. For stand-alone Opal systems, this address is statically determined, and the symbolic name of the backing file is passed to the server on startup. Also supplied at startup is a distinct monotonically increasing value representing the server's *epoch*. The root backing file, address, and epoch are sufficient to recover or rebuild all server state.

The internal server data recovered in the root segment includes embedded rights for Mach kernel objects and ports, including (1) domain tasks, and (2) memory objects for segments attached to these domains and/or to the Opal server itself. The Mach objects, attachments, and ports are invalidated by the shutdown, since the Mach kernel retains no state across restarts. However, the Opal server controls all access paths to these kernel objects, so it can reinstantiate them lazily as resources are called into play by application activity. All port rights in the server's internal data structures are tagged with epoch numbers checked against the current epoch before each kernel call. For example, the memory object right in each *BackingFile* is tagged with an epoch and checked on *Attach*; if the right is left over from a previous epoch, it is refreshed by looking up the file by its symbolic name, which is easily determined from the FileID in the *BackingFile* object.

On recovery, the server's internal data structures are recovered whole by reattaching its segments, specified by a list in the root segment. The internal structures are then "scrubbed" to eliminate records for transient Opal resources. The recovery procedure first walks a list of protection domains that existed at the time of the shutdown, releasing the *Domain* records for all nonpersistent domains, along with their references on attached segments. Next, it scrubs the segment pool(s) that were active at the time of the shutdown, resetting the nonpersistent reference counts on all *SegmentRefs*, and releasing *SegmentRef* and *Segment* objects with no remaining references. Finally, it traverses the resource group tree, and destroys all empty nonpersistent resource groups. Note the wavefront nature of this process: releasing a domain drops its reference counts on attached *SegmentRefs*, which may cause the segments themselves (along with their backing files and address ranges) to be released, and possibly triggering reclaim of emptied resource groups as well.

6.1.7 Discussion

Prototyping the Opal system interface was straightforward given the primitives provided by Mach. The Opal server represents protection domains as Mach tasks, raw addressing contexts whose virtual memory mappings are controlled by the Opal server. Segments are represented as Mach memory objects with address bindings and backing storage controlled by the Opal server. Most details of virtual memory management and backing storage management are handled by the Mach kernel and a backing file server; the Opal server only coordinates their actions. The Opal server implements the global virtual address space by coordinating virtual address usage so that shared segments are mapped at the same addresses in all tasks, and the addresses assigned to different segments are disjoint. Mach supports additional features not needed for Opal, including independent virtual address spaces for each task, copy-on-write memory sharing, kernel-based messaging primitives, and kernel-based capabilities for ports. These could be eliminated in a native Opal kernel.

The Opal server prototype is a proof-of-concept for all aspects of the Opal system, with the exception of low-level memory management for the global address space. In particular, the Mach microkernel for the MIPS and Alpha uses per-task tree-structured linear page tables (Section 3.1.1), which perform poorly for single address space systems under heavy load. A better kernel for Opal would use a combination of data structures similar to those in commercial systems and in the prototype Opal server. The proposed implementation can be defined by its handling of four classes of events:

- **TLB miss.** A global hashed resident page table (HRPT) represents mappings for resident pages, with cached protection keys for domains permitted to access those pages, as described in Section 3.1.2. A large majority of TLB misses would be resolved from this structure with one or two cache line references. The HRPT replaces the current Mach *physical map* layer.
- **Protection miss.** An HRPT probe on a resident page will fail if the page is referenced from a domain whose protection key is not cached in the HRPT entry. In this case, the TLB miss handler follows a back-pointer to the page's segment records, and traverses a list of domains that have attached the segment. If the active protection context is on the list, then that domain's protection key is cached in the HRPT entry. To support per-page permission for certain segments (e.g., to support some form of MMU-based data management [AL91]) attach records on the access list can include a page bit map or some other structure with page-grained access information.
- **Page fault.** If the referenced page is not in the HRPT, the handler probes a splay tree of attached segments keyed by virtual address, similar to the tree used for *AttachByAddress* in the prototype. Microsoft's NT maintains a similar splay tree of mapped regions for each process [Cus93], used for handling some page faults. Information in the segment record can be used to locate a copy of the desired page. In our prototype, these segment records contain a Mach right or NFS handle for the segment's backing file on disk.

- **Page fetch from disk**. Faulted pages are retrieved from their backing files using a per-segment secondary mapping structure of the file server's choosing. The inode pager and NFS server used in our prototype both represent backing files as *inodes* with hierarchical linear block maps. These maps are similar to traditional linear page tables, but note the following differences: (1) they are shared by all users of a segment, (2) they do not include protection state, (3) they are maintained in the storage server rather than in local kernel memory, and (4) each segment has a separate map, which is efficient if the segments themselves are relatively compact, even if they are sparsely placed in the address space.

Because the Opal prototype is implemented as a server rather than a native kernel, multiple address spaces can exist on an Opal host; although programs in the Opal environment share a single address space, programs running in other environments (e.g., Unix) may use address space in a different way. Opal's address space is actually a subrange of the full processor address space, limited to addresses not ordinarily used by Unix processes on Mach. In our prototype, Unix utilities linked with the runtime package can use the Mach name service to bind to the Opal server and access a subset of its facilities, e.g., create Opal protection domains and make RPC calls to them, or create and/or attach segments that do not conflict with addresses used by the Unix process. In fact, our prototype Opal server is itself a Unix process.

6.2 Standard Runtime Package

As stated in Section 6.1.3, our prototype server creates each domain with a pre-initialized instance of the standard runtime package in an initial segment. The runtime system structures in the initial segment include a *MemoryPool* to manage the initial segment as a heap (Section 5.1.1), proxies for the Opal services, a thread scheduler for the threads active in the domain, and support structures for object-based RPC. Previous chapters and sections have covered most aspects of the Opal runtime package. The following subsections review and fill in some gaps for the key pieces of the Opal runtime package: threads, RPC, and the object-oriented C++ features described in Chapter 5.

6.2.1 Threads

Implementation techniques for user-level threads are well-understood and discussed in detail in [ABLL91]. Opal's OThreads thread package implements user-mode threads above Mach kernel threads. It has several Opal-specific features. Each thread descriptor points to a current resource group proxy and a current memory pool for heap allocations; both pointers are inherited from the parent, but are user-settable with runtime system calls. Thread descriptors and thread stacks are allocated from ordinary heap segments. Thread stacks are fixed-size and aligned; the base of each stack holds a pointer to the thread descriptor, accessed by masking bits off the stack pointer. This is used to find the thread's current scheduler on a context switch.

Shared mutexes (Section 5.1.2) are another important Opal-specific feature in OThreads. Shared mutexes can be used for blocking cross-domain synchronization, without significantly impacting the performance of the local case. As with ordinary mutexes, a thread blocks on a *SharedMutex* by linking its descriptor into a waiter queue associated with the lock. However, in this case the queue and links are allocated from the same segment as the lock, so it can be accessed by all users of the lock. In addition to a pointer to its descriptor, a blocking thread also places its domain portal ID and a random key value on the shared waiter queue. When the lock is relinquished, the previous holder compares its domain portal ID with the sleeper's portal ID. If they match, the sleeping thread is awakened in the usual fashion, by placing it on its scheduler's *ready thread list*. If the portal IDs do not match, then the sleeper originated in a different domain; the holder switches through the portal to make a protected call to the sleeping thread's scheduler.

6.2.2 Portals and RPC

Each protection domain in the prototype has a single Mach port for incoming portal switches. The thread scheduler ensures that there is always at least one kernel thread ready to receive messages on this port. Incoming messages on the domain port specify the target portal ID; the runtime system must hash the portal ID to get a record for the target portal, and branch to the portal's entry procedure. This hashing is short-circuited in most cases by caching the last translated portal; it would be entirely unnecessary with native portals.

On the sending side, the runtime system caches mappings from portalIDs to Mach ports. If a thread requests to switch to domain *S* through portal *x* in *S*, the runtime system makes a *ResolvePortal* call to the Opal server, passing portalID *x*, which returns a Mach right to send to the port in domain *S*. Runtime code then caches this association in a local hash table, so it ordinarily resolves each portal at most once. The right for the domain port for a child protection domain is returned as a hidden result from the domain create operation, so there is no need to look up the mapping for freshly created portals in a child domain. The mapping of portals to ports is transparent to user programs, which see only a standard interface for RPC through uniformly named portals.

The runtime RPC implementation is simple because of the guarantees provided by the *mach_msg* primitive for Mach ports [Dra90]. Ports ensure reliable delivery of both the request and the response, and they guarantee that only one response can be sent, and only by the receiver of the call. Opal RPC is implemented with a few hundred lines of code to pack arguments and results into a message, send it, await a response, and return hidden error conditions across the boundary. The thread scheduler may also create or block kernel threads within each domain as processors move between domains for RPC calls and returns. This ensures that some thread is always ready to handle an incoming call, but that kernel threads within a domain do not compete for a processor [FCL93].

6.2.3 Object Environment

Like the low-level runtime system, the object runtime structures described in Chapter 5 are pre-initialized by the Opal server before any application code is attached or executed in the protection domain. The object runtime package is registered with the server at startup. Alternative runtime packages could be installed, but all clients use this package in our prototype.

At initialization, the runtime system is bootstrapped with proxies for the basic Opal services. The Opal server also includes a bootstrap protocol that returns capabilities for service objects on request. Unix processes use this feature to attach to the Opal services through the Mach name server, and initialize an object environment identical to that seen by Opal clients. The Opal server itself uses a configuration of the standard runtime package to serve Opal resources as protected objects, for internal threads and synchronization, and to initialize runtime system structures in the primary data segments of newly created protection domains.

6.3 Prototype Performance

This section reports on the performance of basic Opal primitives, based on our Opal prototype running above Mach (MK83/UK42) on a Digital Alpha processor (DEC 3000/400 AXP, 133.3 Mhz, 74 SPECints). The purpose is to demonstrate that the performance of Opal on top of a conventional microkernel (Mach in this case) is comparable to other environments supported by that kernel, such as Unix.

The cost of certain Opal primitives varies widely because the Opal server preallocates and caches segments and protection domains to meet future needs. Creating a new segment from scratch takes 3.6 *ms* on average, using the Mach inode pager, which performs many directory operations asynchronously. Much of that time goes to creating a segment backing file (inode) for the segment. To reduce this cost, Opal recycles inodes; assuming that the recycle list is non-empty, segment creation time is 315 *us*, which is the time we expect applications to typically see. Performing an attach/detach of an Opal segment takes 478 *us* in the best case. About one fifth of this is the cost of the underlying Mach *vm_map* and *vm_deallocate* operations; the remainder is the cost of two RPC calls to the Opal server, with a small amount of internal processing to allocate address space and record the attachment.

In most cases, protection domains can be created very cheaply, since they are preinitialized and cached by the server. A cached domain can be allocated, called, and destroyed in 3.0 *ms*. The majority of this time is spent destroying the domain; that is, a domain create/call takes about 650 *us* and the destroy a little over 2.3 *ms*. The high destroy penalty reflects the costs of recycling the domain's data segment and tearing down a Mach addressing context. If no cached domains are available, creating a domain from scratch, calling it, and destroying it is substantially more work, requiring 12.13 *ms*. A significant portion of this time is spent zero-filling and initializing runtime system structures, notably the global data block for the standard runtime package; this package includes Mach routines that instantiate several pages worth of private static data, which is expensive on single address space systems (see Section 7.3.1). Still, this time compares favorably with the equivalent *fork/exec/exit/wait* of a null Unix/CThreads program on Mach/Unix (12 *ms*). The underlying Mach/Unix *fork/exec/exit/wait* (without CThreads) is 8.75 *ms.* Note, however, that the same operation on DEC OSF/1 takes only 1.9 *ms* on the same hardware.

This gives some indication of the cost of building any system environment above Mach, but it is not a good measure of the overall performance of Opal. The microkernel structure does not impose significant overhead for common-case cross-domain interactions using RPC or shared memory, once connections have been established. In particular, cross-domain calls do not involve the Opal server once the portal is initially resolved. The base cost for an Opal RPC call is 133 *us*, which is in the same range as cross-address space calls on Mach (88 *us*) and other systems [BALL90]. This includes the cost incurred by our runtime support for password capabilities built above Mach RPC.

Implementing the Opal kernel as a Mach server clearly entails some performance loss compared to a kernel-level implementation (as it does for Unix as well), and we can identify the major costs.

- The Mach virtual memory system was designed for private address spaces, with a separate translation map for each task. This structure incurs significant overheads for handling sparse virtual addressing in an Opal system. In particular, the costs of recycling (invalidating) segments and of tearing down addressing contexts grows very rapidly with the amount of virtual address space used. We have not reported these costs, as the problem and solution are well understood (see Section 3.1 and Section 6.1.7).
- Our unmodified Mach kernel does not support native portals. Whether or not this impacts the common-case latency of raw RPC, it adds hashing overheads on the sending side and sometimes the receiving side of our user-level RPC package. In addition, given the explicit reception model of Mach messages, the user-level thread scheduler must occasionally create additional kernel threads to listen on the domain port. For example, creation of an initial listener thread represents roughly 25% of the cost of preparing a new protection domain.

- All Opal memory segments are potentially sharable and persistent. In Mach, this requires that they be backed by an external paging server, which makes zero-fill page faults (336 *u*s each) and segment deletes (900 *u*s for standard memory pools), somewhat more expensive than the "temporary" virtual memory used by Unix on Mach. We believe that this cost could be reduced by improving the kernel-pager interface.
- The Mach thread interface is the basis for our user-mode thread package, rather than scheduler activations. This impacts performance in an environment with frequent cross-domain communication. Some of the implications are discussed in [FCL93], which reports additional performance results for threads, RPC, and cross-domain synchronization in the Opal prototype.

Overall, the points to be gleaned from our Mach experience and performance data are: (1) the Opal primitives have performance that is reasonable in an absolute sense, and (2) they do not add significantly to the cost of the underlying Mach abstractions on which they are implemented. There are no significant hidden costs to the single address space model. Although an improved kernel implementation would yield better performance for our prototype, preallocation and caching of Opal resources negates most of the microkernel costs in the common case. For example, the cost of allocating an Opal protection domain is a small multiple of the base RPC cost, and significantly cheaper than creating a process in an optimized commercial Unix kernel. In any case, we expect complex applications built for the Opal environment to ultimately perform better due to the ease of direct memory sharing and the simplified use of persistent storage, as illustrated in the next section.

6.4 Structuring Opal Applications Using Mediators

This section presents a simple shared memory program as an example of application use of the Opal facilities. The experiment implements a variant of a tool integration facility called *mediators* as an application-level server above the Opal facility. The Opal-based mediator facility is intended to illustrate a useful paradigm for structuring tool relationships in an integrated application environment such as the Boeing CAD system described in Section 2.1.

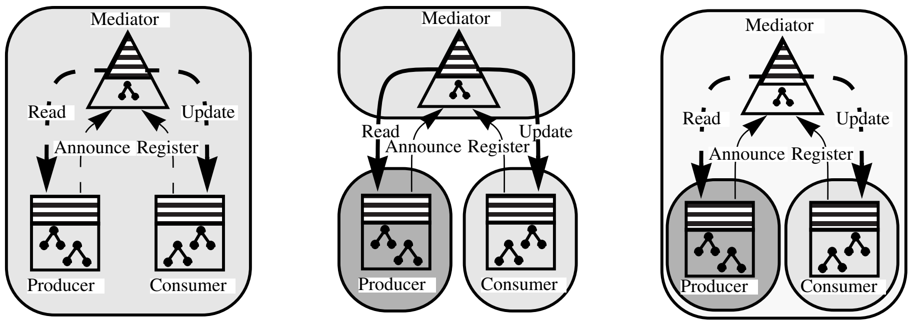
This Boeing CAD system shares a key problem with other integrated design environments: given a shared database containing source objects, how do we efficiently propagate source updates to other tools that maintain local structures derived from those sources? In the Boeing environment, the source objects are aircraft part records, and the derived objects belong to tools doing simulation, spatial analysis, and so on. With Opal, we wish to access source parts in shared memory, updating the derived objects *incrementally* rather than rebuilding them, due to their large size.

This approach can significantly improve performance and scalability. However, integrating independent tools in this way can be quite complex. Tools must make assumptions about the format of their input and adhere to standards for the format of their output. These standards are necessary regardless of how the tools communicate, but the richness and fragility of shared pointer structures would seem to force compromises in the independence of the tools. In this section we show that software structuring techniques exist that can facilitate the fine-grained interactions needed for incremental updates to large information structures, without compromising modularity or component independence.

The framework is based on a variant of a *mediator* facility developed by software engineering researchers [SN92]. Mediators do not solve the integration problem, but they do reduce its complexity. The idea behind mediators is that knowledge of inter-component dependencies is factored out of the components themselves and into separate components -- mediators -- that coordinate the behaviors of the components. This integrates the components while allowing them to evolve independently. components communicate implicitly by announcing *events* when updates occur. Events announced by data-producing components are delivered to mediators acting on behalf of data-consuming components; these mediators then make a series of calls to both components to propagate the updates. components and mediators should be thought of as modules with private data, executing in a variety of possible protection configurations. The calls to announce and propagate updates may be either ordinary procedure calls or protected procedure calls, depending on the protection relationships among the components involved. This structure is depicted in Figure 6-3.

Our Opal mediator package includes runtime support and a protected *event manager.* components register with the event manager, specifying the events they announce, along with capabilities for segments containing procedures and data related to those events. Mediators register with the event manager, specifying events they wish to receive. The event manager sets up the appropriate RPC connections and segment attachments among components and mediators. Our mediator implementation is a test system built for demonstration purposes, however, the mediator paradigm has been used to implement systems for program restructuring [GN93], computer-aided geometric design [McC91], and radiotherapy planning [SKN95], among others.

Opal applications built using mediators can be *transparently* configured to use different protection arrangements, including overlapping protection domains and shared memory. To illustrate the structuring options and their performance, we implemented a simple mediator-based application. A source tool (the *producer*) repeatedly creates and deletes fixed-size records, maintaining a tree index on those records and announcing events after inserts or deletes. A derived tool (the *consumer*) maintains and uses its own index structure on the same data, while a mediator keeps the derived index up to date in response of changes by the producer.



Monolithic Configuration

A mediator propagates updates from the producer of a data structure to a consumer maintaining a derived structure. All components execute in a single domain ("fast").

Decomposed Configuration

All components execute in different domains ("safe"). Data returned by ***Read****() is linearized in a reply message. In our experiments, the consumer and mediator actually share a domain.*

Overlapping Configuration

The mediator's domain has read access to the producer and read-write access to the consumer. The mediator transfers data through shared memory ("fast and safe").

Figure 6-3: Three configurations of a tree-indexing program using mediators.

An asymmetric trust relationship exists between the tools. The producer can be isolated from both the mediator and the consumer, but both of them trust the producer to supply well-formed input data. Figure 6-4 shows the performance of this application for three protection configurations, as a function of the number of records processed:

- *Monolithic*. The producer, consumer, and mediator share a protection domain. This configuration occurs in Unix when components are linked into a single image and execute as a single process.
- *Decomposed*. There are two non-overlapping domains: the producer in one, the consumer and mediator in the other, communicating with the producer by RPC. A similar configuration exists in Unix when tools execute as separate processes (sacrificing performance).
- *Overlapping*. There are three Opal domains. The mediator domain has the producer data and code segments attached read-only, and the consumer segments attached read-write. When the producer announces an event, the mediator is invoked with an RPC call, passing the address of the modified elements in the shared segment.

Figure 6-4 confirms the performance benefits of sharing for this environment. In this example, the performance of the decomposed configuration is determined by (1) the cost of copying data across the protection boundary, and (2) the costs of rebuilding heap-allocated data structures in the consumer (synchronization costs are not included). The overlapping organization, while as safe as the fully decomposed configuration, is nearly as fast as the monolithic configuration. Furthermore, the performance advantage relative to the decomposed configuration increases as the amount of shared data increases.

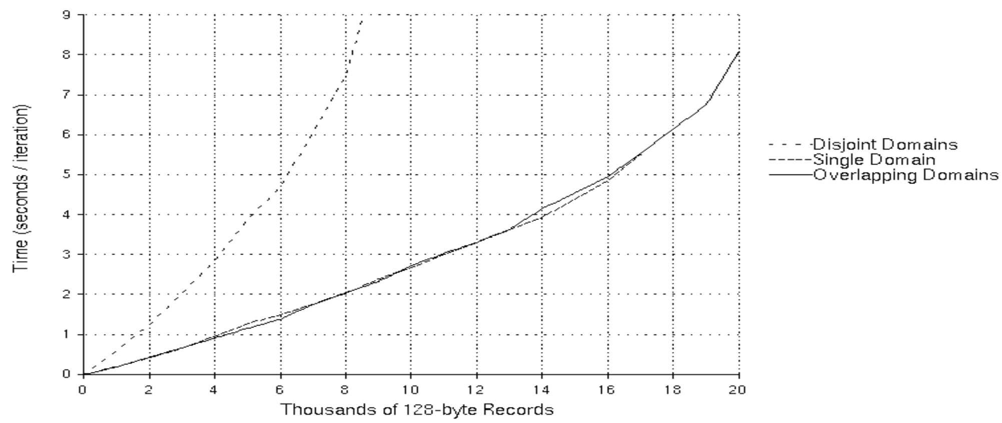


Figure 6-4: Execution time for the mediator-based tree-indexing program.

The key points of this example are that: (1) read-only shared memory is a natural and efficient alternative to pipes, files, or RPC, (2) read-only shared memory preserves the one-way isolation provided by pipes and scratch files, and (3) programs structured using mediators are easily adapted to run in overlapping protection domains. In fact, all three configurations of our tree indexing program are built from the same source code; the shared memory is completely invisible to the program. In addition, there is a complete separation of modularity and protection; all interactions are through procedure calls and clean modular interfaces. Overlapping protection domains exist ‘‘over top of’’ these interfaces, representing different trust relationships. For example, an event signal (from producer to mediator) crosses both module and protection boundaries; an update call (from mediator to consumer) crosses module and *possibly* protection boundaries; a source object read call (from mediator or consumer to producer) crosses module but not protection boundaries.

6.5 Summary

This chapter described our Mach-based Opal prototype, which demonstrates that a single address space environment can be implemented in a straightforward way on a microkernel operating system. The Opal environment coexists with other operating system “personalities,” notably Unix, thus permitting use of existing Unix applications to support and monitor the Opal programming environment. This implementation strategy enabled a pair of full-time graduate students and collaborators to implement a skeletal Opal prototype in under a year.

The Opal prototype has been used as a basis for experiments with (1) a mediator-based paradigm for integrated systems, (2) alternative approaches for supporting a pointer-based CAD database [Nar95], and (3) shared memory optimizations for read-mostly RPC services, such as name services [Sch94]. This chapter described the mediator experiment, which illustrates how shared memory and overlapping protection could be used to transparently improve performance or reliability of integrated systems constructed using appropriate software engineering methodologies. In particular, this technique opens opportunities for incremental propagation of updates through related data structures maintained by cooperating applications, which we believe is a promising approach to improving the scalability of integrated systems such as Boeing's aircraft CAD environment.

Chapter 7

Preparing Executable Code

This chapter explores the issue of preparing user code to execute in a single virtual address space operating system. In particular, it discusses *linking* -- representing and resolving internal program references in executable code -- in the Opal prototypes for the 32-bit MIPS and 64-bit Alpha processors. Opal's addressing model presents both opportunities and challenges for the linking system. On the one hand, code and data are easily shared by reference in the global address space. On the other hand, context-dependent absolute addressing cannot be used, which can complicate the handling of private data. Our discussion of code handling illustrates the key issues for data management in a single address space.

The fundamentals of compiling and linking are unchanged by the single address space. In fact, the Opal prototype uses standard freeware compilers for the C and C++ languages (**gcc** and **g++** from the Free Software Foundation), and linkers supplied for private address space systems by the processor vendors. This chapter is concerned with additional steps to prepare code for the single address space *after* it is processed by the standard linker.

In many respects, the issues for handling code in the single address space are the same as for data. Like data, executable code contains embedded references, in this case to branch targets and global data operands. The addressing model shapes how these internal references are represented, particularly for data or code that is shared among addressing contexts or dynamically imported (loaded) into running programs. In the case of executable code, support for dynamic sharing of procedures is an important goal. In an operating system with multiple addressing contexts, sharing reduces memory consumption and context startup costs; dynamic loading permits a program to change its behavior or extend its functionality by importing new procedures at runtime, without the need to create a new addressing context in which to execute them. This chapter develops linking facilities to support dynamically shared procedures in Opal.

We propose a new linking technique called *global static linking*, made possible by Opal's single virtual address space and support for persistent segments. The defining characteristic of global static linking is that shared symbols, including pure procedures, have a fixed location in the address space, and are referenced with statically determined virtual addresses that can be stored, passed, and shared. Executable code exists within segments; procedures are dynamically imported by attaching segments, and shared by sharing segments. Any procedure can in principle be shared or called at any time by any thread with a pointer and authority to attach the procedure's segment.

Global static linking in Opal enables a degree of flexibility that requires more expensive dynamic linking in private address space systems. In addition, procedure pointers are globally meaningful. However, the single address space complicates the handling of *private* (context-dependent) linkages in shared code (e.g., for private global data symbols) and private changes to shared code (such as setting a breakpoint). We discuss linking in detail because it illustrates the basic themes of this dissertation: the promise of simplified sharing in a single address space, and the issues that must be resolved in order to deliver on that promise in practice.

7.1 Background and Terminology

This section introduces some important terms and concepts for the discussion that follows. Some of the terminology is derived from Unix-like systems, but the concepts are generally common to all systems, independent of the programming language and addressing model. The purpose of this section is to introduce the basic process and phases of linking. The next section outlines the effect of the single address space on these phases.

The inputs to the process are *code objects* produced by source language compilers and assemblers. Compilers segregate the procedures and data items defined in the source code into different *sections* in each code object, depending on how those items are used at runtime. For example, the assembled instructions are placed in the *text section*. In addition, each code object may contain one or more *data sections* of different flavors (readonly, writable, packed short, uninitialized or **BSS**, and so on) containing constants or initial values of static data items defined in the source code. A data item is *static* if its storage is allocated at a fixed location before the code executes, rather than at runtime from a stack or heap. A static data item is *private* if its storage can be modified independently by different callers of the module. Private static data items are typically global variables, an important but troublesome feature of many popular programming languages including C.

In addition to the various sections, the code object contains a *symbol table* listing the names and locations of (1) externally visible code symbols and global data symbols defined in the object, and (2) external symbols imported from other code objects. Code objects may be collected in *archives* with a common symbol table.

A code object is the simplest form of *module*, the basic unit of input to the remaining linking and execution steps. A module may consist of several code objects statically combined by copying and merging their sections and symbol tables into a single output object. This combining step also involves other steps explained below. A module may be called a library, package, program image, load image, or "a.out file", depending on its contents and function.

Modules undergo four distinct processing steps before they execute. These steps are called by various names and are easily confused. Here are my definitions.

- **Linking**. The symbol tables of a group of modules are processed to resolve references within each module to external symbols imported from other modules.
- **Binding**. The addresses at which the module's code and data sections will reside at execution time are determined.
- **Relocation**. The values of certain fields within each section are updated to reflect the new location of some symbol, determined from a previous link step or bind step. For example, the text section contains memory reference instructions (loads, stores, and branches) with literal fields encoding the addresses or offsets of their operands. Similarly, the initial value of an item in a data section may be derived from the address of another symbol. Each section contains a table of *relocation records* identifying these fields and the symbol table entry for the item each field depends on.
- **Instantiation**. The sections of the module are loaded (or mapped) into memory for execution.[1] Sections are page-aligned, and may be mapped in different modes. For example, readonly sections might be mapped readonly; text sections are mapped with *execute* mode if required by the processor. The instantiation step also allocates and initializes storage for private static data items defined in the module.

These steps may occur at different times in different systems, or even within the same system, since the constituent parts of a program may be constructed in different ways. In general, each of these steps can be

1. Instantiation is sometimes called *loading*. I use a different term because loading is confused with linking in systems that bind all load addresses at static link time. For example, the Unix static linker is often called the "loader" or **ld**.

completed statically or dynamically, defining a continuum of linking systems ranging from fast and inflexible to slow and flexible. Private address space systems support three basic ways to link modules together to construct programs.

- **Static link-and-load**. The sections of the source modules are gathered together, merged, and copied into a single output module, with all internal references linked and relocated. In older Unix systems, the result is typically a complete, self-contained executable program image that is statically bound to execute in a private virtual address space; the linker assigns the same starting address to every program. The advantages of this style are that the image has few dependencies on its environment, and global symbol references can be optimized at link time since all load addresses are known and the entire program is present. The disadvantages of this style are: (1) the entire image is copied, (2) procedures and data items cannot be shared with other images, and (3) new versions of the image's consituent parts are not incorporated without explicit relinking.
- **Dynamic link libraries (DLLs)**. DLLs are bound and instantiated as needed, and references pointing into them are resolved lazily. The primary advantages of DLLs are: (1) they need not be replicated in each program's load image, saving space on disk, (2) new versions can be automatically incorporated without rebuilding the calling modules, and (3) they are typically shared at runtime, improving virtual memory performance. However, DLLs cost more to instantiate, and opportunities for link-time optimizations may be lost, leading to less efficient (but more general) sequences for referencing global symbols.
- **Runtime linking**. Procedures are called through a procedure pointer at runtime, typically with an extra address load from a procedure vector that is statically or dynamically linked. Runtime linking is flexible because a given call site may be used to call different implementations of the same interface at runtime.

In some systems, constructing a program can involve a combination of all three styles. For example, a programmer using Digital's OSF/1 system for the Alpha can create a statically linked image that imports a DLL at runtime, then calls procedures in both modules as runtime-linked virtual functions.

7.2 Global Static Linking for a Single Address Space

Use of a single address space affects code handling in two fundamental ways, stemming from the manner in which the virtual address space is allocated. On the positive side, shared symbols (e.g., text and readonly data symbols) always reside at common virtual addresses, simplifying sharing. However, separate instantiations of private symbols *must* reside at *different* virtual addresses. This complicates sharing of modules that reference private static data. Moreover, the address range assigned to a new segment cannot be known in advance, since the address space serves all users. If we suppose that the segment for each module is allocated only when the module is needed at runtime, instantiating a module always requires some form of dynamic relocation, even for shared symbols. For this reason, earlier single address space systems, including Pilot and Cedar (Section 2.2.1), and Psyche, use dynamic linking exclusively.

The 64-bit virtual address space assumed for Opal is large enough to reserve virtual address ranges for segments when they are constructed, and retain them even when they are not in active use. This presents a new opportunity to *statically* link modules into the single address space. We have experimented with this approach, called *global static linking*, using our prototype interface for allocating persistent virtual memory segments. Global static linking supports dynamically shared modules without the costs of dynamic linking, and it enables pointer-based code binding for shared and persistent objects (see Section 5.1.3). However, it raises several concerns that we have not fully resolved in our prototype.

This section discusses the ideas behind global static linking, its implementation on the MIPS and Alpha platforms, the issues raised by this approach, and some proposals for dealing with those issues. To simplify the discussion, suppose for now that all modules are *chaste* modules in which all writable data items are allocated from the stack or the heap.[1] The issue of private static data is deferred to Section 7.3.

7.2.1 Binding and Resolving Globally Linked Modules

In the Opal prototype, modules are prepared by a collection of standard Unix tools coordinated by a build script called **OpalLd** that coordinates their actions. **OpalLd** takes the following actions for each module.

- Allocate a segment for the module from the Opal address space manager, using the *ReserveModule* interface that allocates an unbacked segment of virtual address space.
- Invoke the Unix static linker **ld** to (1) gather a collection of code object files into a demand-paged shared text (“OMAGIC”) load module using static link-and-load, (2) bind the text and readonly data symbols defined in the module to the virtual address range assigned by *ReserveModule*, and (3) resolve and relocate all intra-module references.

1. The term *chaste* was suggested by John Wilkes, as the related terms *pure*, *clean*, and *reentrant* do not quite capture the concept. A chaste module must be a *pure* module with no self-modifying code; impure modules cannot be shared in Opal. In addition, a chaste module must be *clean*, meaning that it has no references to writable private static data (i.e., no non-shared global variables). Note that chastity is neither necessary nor sufficient for a module to be reentrant.

- Install the new load module as the backing file for the module's segment, using the address space manager's *FillModule* entry point.

At this point, the code in the newly created module is executable, but it may contain unresolved references to external symbols defined in other modules. The module includes all symbol tables and relocation records, for use in resolving these references. A new utility called **resolve** links external symbol references in the (target) module to definitions of those symbols in other (source) modules that have been prepared in a similar manner. The **resolve** utility patches cross-module references (e.g., calls to the standard runtime package), resolving symbols imported from a list of modules passed as an argument.

Multiple definitions of a symbol may exist in different modules. Name scoping for external references is determined by the ordering of the arguments passed to **resolve** by the build utility (e.g., **make**). Note that **resolve** makes only one pass over the sources, never modifies the sources, and never copies or coalesces sections or symbol tables; it only fills in symbol table entries for symbols imported into the target module, and relocates fields in other sections of the target module, as indicated by relocation records referencing the modified symbol table entries. Mutually recursive modules require two runs of **resolve**, specifying each module as the target.

In this way, Opal modules are statically linked in persistent segments of the global address space. Intra-module and inter-module references to text and read-only data are represented as statically bound addresses. This global static linking has three distinct benefits:

- Threads executing in a module can easily share a single physical copy, even if they are running in different protection domains. All linkages to shared symbols are globally meaningful without dynamic linking.
- Pointers to global symbols can be freely passed and shared at the system level. For example, software components in some protection domain might acquire procedure pointers at runtime, either in shared memory or in a received message.
- Any thread can dynamically attach and call any accessible procedure, simply by knowing its address, with no possibility of address conflicts with other code attached to the thread's protection domain. Modules can be instantiated in any context, because each module resides at a fixed address, the same address in any domain.

Any globally linked chaste module can be used as a dynamically imported shared library in Opal: any thread can invoke its procedures even if it was written and compiled after the thread's protection domain was activated. Furthermore, any procedure can be executed as a "program"; that is, it may be wrapped in a private protection domain to prevent it from accessing the caller's data, and invoked through a portal, as described in Chapter 5. The concept of a *module* encompasses the concepts of a program and a library.

Global static linking is used in a limited way in some dynamic linking facilities. For example, Digital's OSF/1 for the Alpha provides a linking facility called Quickstart [Dig92b] to reduce the instantiation cost for DLLs by retaining linkages from the last instantiation. DLLs are linked at fixed locations, loosely coordinated by an (optional) system-wide registry of allocated locations. When a previously linked DLL is imported at runtime, the OSF/1 runtime loader attempts to place it at its previously linked address range. If the desired range happens to be available in the process address space, then dynamic linking is avoided. Global static linking essentially extends this scheme to encompass all modules in the system; the single virtual address space guarantees that the allocated ranges are always available in the target context at runtime.

7.2.2 Representing Linkages on 32-bit and 64-bit RISCs

Since the symbols imported by **resolve** are statically bound in the source modules, the cross-module references in the target can be resolved to absolute addresses, just like the intra-module references. This is achieved somewhat differently on the 32-bit MIPS R3000 and the 64-bit Alpha AXP, the two platforms used in our experiments.

On the MIPS, the operand addresses for targets of branches to global text symbols and (less frequently) references to global data symbols are formed from absolute addresses coded as pairs of 16-bit immediate operands. The code uses a pair of instructions, **lui** and **addiu**, to build a 32-bit global symbol pointer in a register. The instructions and sequences used to form global symbol operand addresses from a series of 16-bit immediate fields are given in Table 7-1. Note that neither **lui** or **addiu** references memory; each pair completes in two cycles. On this architecture, **resolve** patches pairs of 16-bit immediate fields directly in the instruction stream.

The 64-bit Alpha AXP uses similar instructions and instruction formats: like the 32-bit MIPS, the Alpha has 32-bit instructions with 16-bit immediate operands. However, the Alpha differs in one critical respect: 64-bit addressing increases the number of cycles needed to load a pointer operand from immediate fields in the instruction stream. As shown in Table 7-1, loading a 64-bit immediate operand on the Alpha requires two **ldah-lda** pairs with an intervening 32-bit left shift, consuming five cycles instead of two (note that dependencies between these instructions defeat dual-issue on the Alpha).

The consequence is that code emitted by typical compilers represent global symbol references quite differently on the Alpha: all global symbols are addressed indirectly by loading the operand address from a *glo-*

Table 7-1: Loading immediate addresses on the 32-bit MIPS and 64-bit Alpha.

31	15	0
REFHI	REFLO	

```
lui     r0, <REFHI>
addiu   r0, r0, <REFLO>
movl    r0, 0(r0)
```

63		31	0
WORD3	WORD2	WORD1	WORD0

```
ldah r0, r0, <word3>
lda  r0, r0, <word2>
sll  r0, 32, r0
ldah r0, r0, <word1>
lda  r0, r0, <word0>
ldq  r0, 0(r0)
```

bal address table (GAT). For statically linked modules the addresses in the GAT are computed statically and stored in the *literal address* (LITA) section of the module. Each module's LITA section contains entries for both imported symbols and symbols defined locally. Global static linking on the Alpha involves cross-linking the LITA sections of different modules; **resolve** does not patch instructions, which are position-independent. Instead, it fills in unresolved LITA entries for text and readonly data symbols imported by the target module with the values in the corresponding LITA entries of the source modules.

7.2.3 Portability of Executables

One issue for global static linking is that it must be possible to install "shrink-wrapped" executable software, distributed in binary form, into a single address space system using global static linking. This is easily achieved if the installation process includes a binding and relocation step supported by the binary format. For self-contained binaries on the Alpha and similar 64-bit machines, this simply involves relocating the addresses in the LITA array by a constant offset; no symbol table or relocation table is needed. However, if the code consists of multiple modules, or contains unresolved references, then the symbol tables and relocation information must be included, along with a sequence of source and target lists for **resolve** to link the installed modules into the target system.

7.2.4 Evolution

Another issue for global static linking is incorporating linkages to new versions of modules. When a module *M* is modified, the new version *M*' is linked into a different part of the address space; the old version *M* continues to exist at its assigned addresses. Other modules may hold static linkages to symbols in *M*, which continue to be valid. The programmer may want these references to rebind to *M*', but as always with static linking, this does not occur automatically. In our prototype, the programmer must issue an explicit **resolve** command to rebind, typically with a build utility that records dependencies (e.g., **make**).

If the programmer truly wants to select the module version at instantiation time, then the library must be dynamically linked.

There are several possibilities for automatically incorporating new versions of globally linked modules. One proposal is for **resolve** to record source module dependencies in a *fan-out table* in each target module. A *linkage server* examines the table each time an inactive module is instantiated, and re-resolves links to any changed modules in its fan-out. In fact, OSF/1's Quickstart optimization takes a similar approach: the dynamic linker decides at runtime if dynamic linking is needed, in part by checking for modified target modules. One difference is that the OSF/1 dynamic linker operates on a process-private copy of the module's GAT, whereas the proposed linkage server modifies a shared system-wide copy of each GAT, and must be privileged.

7.2.5 Reclamation

A third issue for global static linking is that the system must not reclaim modules while other modules have static linkages into them. Statically linked callers of a module hold linkages to a *particular version* of a module; the callers must be relinked before that version can be reclaimed. If that version is reclaimed prematurely, its callers will fail to execute correctly. Dangling references are also a concern for many DLL facilities, in which callers may specify a particular version of the library, or expect to locate it with a particular symbolic name.

Old versions of modules are reclaimed manually in our Opal prototype. When a new version *M'* of a module is linked and entered into the name server, it takes precedence over older versions *M*; a *LookupModule* call on the module's symbolic name returns the most recent version. However, the name server does not yet release the reference count on *M*, so other modules that reference it can still attach it by virtual address. A *PurgeModule* operation and utility are used to release all references held by the name server on previous versions of a named module.

This manual reclamation is error-prone because there is no way to determine if other modules are linked to versions that are candidates for destruction. One possibility is to extend **resolve** and the proposed linkage server to maintain a *fan-in* table as well as a fan-out table in each module. The fan-in table in module *M* contains pointers to modules with statically linked references into *M*; the fan-out table contains pointers to all modules that define symbols imported by *M*. When *M* is destroyed, the linkage server visits each of the target modules listed in *M*'s fan-out table, and removes the entry for *M* in each target module's fan-in table. In fact, the Gnu dynamic linker **dld** [HO91] uses a similar reference-counting scheme to track references to libraries loaded and unloaded into the virtual memory of an executing Unix process. The key difference in this case is that we are modifying the *permanent* linked image rather than a transient copy. This means that updates to fan tables and reference counts must be atomic with respect to failures.

7.2.6 Effect on Link-Time Optimizations

Global static linking may sacrifice opportunities for some link-time optimizations developed for static link-and-load in private address space systems [SW94]. In particular, any optimization that relies on knowing all call sites for a procedure cannot be applied to procedures exported from a module, unless all importers are known to the optimizing linker. Most link-time optimizations can be applied to a closed group of modules, but not to modules containing public procedures in the global address space.

Some optimizations rely on short jumps. Cross-module jumps may be farther apart in a single address system, since addresses are assigned to modules at the system's convenience. However, note that the system does not impinge on the right of programmers to keep multiple copies of the same procedure linked in different parts of the address space, to improve code locality or cache performance at the price of consuming additional memory. A programmer applies these optimizations by constructing private modules with copies of procedures drawn from standard archives, rather than linking to the public copies directly.

7.2.7 Summary

This section discussed some issues for linking in a single address space, and presented a new approach, global static linking, for Opal's persistent shared address space. Global static linking enables efficient and flexible dynamic sharing of procedures. However, like all statically linked code, globally linked modules do not automatically incorporate new versions of their procedures. Like any linked persistent structure, some care is needed to safely reclaim from the web of statically interconnected modules. Finally, modules imported from outside of the address space must be linked into it before they can be used. I have proposed solutions for each of these concerns, but our prototype uses less transparent mechanisms.

7.3 Handling Private Linkages

The single address space complicates handling of private static data, which can have multiple private instances addressed from shared code. To this point we have assumed that modules are "chaste", meaning that all static references are shared; the module does not define or reference any items that can be modified independently by different users of the module. Our prototype imposes limits on the number of non-chaste or "dirty" modules that can be attached to a given protection domain, as we shall see.

The basic problem is that private symbols in a single address space system can *never* be statically bound. Different instances of these symbols must exist at *different* virtual addresses, which cannot be determined until instantiation time. This means that references to private static data in shared code *must* be position-independent; absolute addresses cannot be used. PC-relative modes are not useful either in this case, since any shared code must by definition reside at the same addresses for each instance, so the PC offsets will be different for each caller as well.

Instead, instructions must address private symbols as offsets from a designated base register, the *global pointer* or **gp** register, with a different base value for each instance. This sort of register-relative addressing is a given on RISC processors, particularly wide-address processors. Code generated by standard compilers for both the MIPS and Alpha already use **gp**-relative addressing extensively for performance reasons, since the offsets can be encoded as immediate operands in the instruction stream. The real problem is that single address space structure imposes new constraints on how the **gp** base values are determined for shared modules, *if* (but only if) private data is used. In particular, the standard calling conventions must be changed.

The standard conventions for Ultrix/MIPS support only static link-and-load of traditional Unix: only one module can be active in each process; all global data is placed in a single global block, with nonconflicting **gp** offsets assigned by the static linker. This is easy to emulate, but it is not very interesting. Thus we consider the Alpha as an example.

To support dynamic module sharing on the Alpha, each module must have its own global data block, addressed through its own global pointer base value. Under OSF/1, global data and text symbols are addressed indirectly through the referencing module's global address table (GAT), which is itself addressed relative to the global pointer. Standard instruction sequences emitted by the compiler deposit the module's GAT address in the **gp** register each time control switches into the module. The module **gp** values are determined from the target PC on each cross-module call and return. Like other high-performance RISC architectures, the Alpha is deeply pipelined and does not provide a software-readable PC register; instead, the calling conventions dictate that the call or return address is always available in a designated general-purpose register. The module's **gp** value is computed on entry to each procedure as a static literal offset from the call address, and restored after each return as a literal offset from the return address. Thus static data addressing is ultimately PC-relative on the Alpha.

For Opal on the Alpha, all callers of chaste modules address a single copy of the code and its GAT at fixed addresses. The load address of the GAT is statically bound as a constant offset from the shared code using the standard Alpha calling conventions. This works because the GAT contains addresses for shared symbols only, which are statically bound and fixed for all callers of the module; thus the GAT itself is immutable and can be shared.

Suppose, however, that the module is not chaste, that is, its GAT contains an address for a private symbol. Each instance of the module must address a private copy of the private symbol at a different virtual address. Thus the symbol address in the GAT entry must have a different value for different instances of the module. This has three troublesome implications:

- Since the GAT entry values differ and the GAT must be stored contiguously, the entire GAT must itself be treated as private static data, even though it is not modified by the program as it executes.
- If different instances of the GAT are private, the constraints of the single address space dictate that they must reside at different addresses, like any private static data. Thus a new copy of the GAT must be allocated, copied, and relocated by a runtime loader each time the module is instantiated.
- The module's global pointer base value -- which points into the GAT -- can no longer be computed PC-relative as a literal offset from the text address (unless the text is copied and relocated as well). Thus the **gp** must be saved and restored from memory on cross-module calls. Section 7.3.3 presents some proposals for implementing **gp** switching.

OSF/1 and other private address space systems avoid these difficulties with static data by overloading virtual addresses. Each instance of a given module exists in a separate virtual address space; the private copies of its global data typically reside at the *same* virtual addresses within those separate address spaces, although they map to different physical locations for the different copies. For example, several Unix processes executing a shared C image typically reference the program's global variables using the same statically determined addresses.

In fact, most modern Unix systems (including OSF/1) defer the cost of copying the multiple instances with copy-on-write. That is, multiple instances share a copy of the module's initial data, until one of them writes a page, at which time the MMU raises an exception, prompting the supervisor to copy the page and remap it on the fly. Copy-on-write increases the cost of private static data, but the cost is incurred only if the data is actually modified. In a single address space system, the runtime loader must relocate initial values of internal references that point into the private sections. In general, this includes the value of any symbol (i.e., in a data section) whose value derives from the address of a private symbol; these are statically determined constants under OSF/1. (Note that this is independent of whether or not the data is addressed indirectly through a GAT.) This generally precludes lazy instantiation with copy-on-write, as explained in Section 3.3.4. In Opal, private sections are copied and relocated by a runtime loader at instantiation time.

7.3.1 Global Data in the Opal Prototype

Our prototype treats each attachment of a module as a separate instance, with a private copy of any global data. Each domain has exactly one instance of every module it attaches, and two domains never share an instance. This is a simplifying assumption in this section, and a policy in our prototype. Note, however, that if the linking conventions support multiple instances in a single address space, then the presence of protection domains is of no consequence, since the **gp** base register value of an executing thread alone determines the instance that thread addresses. Threads in multiple domains can share an instance, or multiple instances can be called in the same domain, as determined by the language environment.

The prototype imposes a basic restriction on the use of non-chaste modules. Each protection domain may attach only two modules that are not chaste: a "standard" runtime package, and one application module of its choice. Note that this is essentially a restriction on granularity: programs may use global variables freely, but if they do, the entire program must be linked into a single self-contained module, just as in Ultrix and other Unix-like systems based on static link-and-load. Our application model is that one module in each domain defines private global variables that point to the "roots" of data structures used from that domain, which are heap-allocated and possibly shared; other modules will be chaste modules attached as needed to operate on data structures of particular types.

Opal uses a single global pointer base value for both of the dirty modules called from each domain. The instructions in dirty modules are statically processed by a utility called **purify** to remove any code that modifies the **gp** register, and coordinate **gp** offsets among these dirty modules. Dirty application modules are assigned offsets that do not conflict with those assigned to any standard module, hence any dirty module may share a global block with a standard module. However, **purify** assigns conflicting offsets to different non-standard dirty modules, so only one such module can be attached to each domain.

As explained in Section 6.1.3, each newly created protection domain is assigned a default segment in the Opal prototype. A contiguous 64K *global block* is allocated from this segment to store the private static data of modules called from that domain, addressed from a single global pointer base value. When a dirty module is instantiated, the runtime loader copies all writable data sections (including the LITA section for the GAT) from the module image into the domain's global block at the offsets assigned by **purify**, then it relocates any value initialized to the address of another private static item.

On the MIPS, the dirty global pointer is the only **gp** value ever used by the domain; the **gp** register of each thread entering the domain through a portal is initialized to this value. In the dirty modules, all private data references are relative to this **gp**. The chaste modules contain no **gp**-relative references of any kind; the value of the **gp** is irrelevant. The primary role of **purify** is to convert any absolute private references into

gp-relative references. The compiler may use absolute modes (e.g., for "large" data items such as private static arrays) in order to conserve space in the global block; thus **purify** may cause offset space to be consumed faster. On the MIPS, under either Ultrix or Opal, linking fails for modules that exhaust their offset space. However, under Ultrix, absolute modes may be used to conserve offset space with some performance cost; this option is not available for shared modules under Opal.

On the Alpha, all symbols used by a module are addressed through that module's GAT. The GATs for different modules used by a domain are addressed through the **gp**; code sequences for cross-module calls and returns may change the value in the **gp** to switch to the active module's GAT. As on the MIPS, dirty modules and standard modules share a single GAT addressed at coordinated offsets through a single **gp** value. Chaste modules change the **gp** to point into the shared GAT for the module, using the standard conventions. The basic scheme is that **purify** nulls out the standard **gp** switching sequences in dirty modules, replacing the sequences for outgoing calls to save the **gp** on the stack, and restore it after the return. In the case of cross-module private data references, both the source and target of the reference are dirty modules, and therefore they will share a global pointer at runtime; **resolve** must patch instructions that reference private data to use the same GAT offset used by the module that defines the symbol.

At present, our Alpha prototype has a severe limitation: a chaste module cannot reference *any* symbol defined by a dirty module. Procedure references are not permitted because the **gp** of the callee cannot be determined. Global data references are not permitted because a chaste module does not share a global pointer with the module that defines the symbol, even if that module is the standard runtime package. In practice, this prohibits calls to *printf*. These restrictions can be eliminated if each domain's dirty global pointer is placed at a fixed offset from the stack base of each of its threads, as described in Section 7.3.3, but this has not been implemented.

7.3.2 Compiler Interactions

In theory, our assumption of chaste modules is reasonable for some programming styles that shun global data, particularly the object-oriented style advocated in Chapter 5. In practice, standard compilers on the MIPS rarely if ever produce chaste modules, even for programs with no global variables. Chaste modules are somewhat easier to obtain on the Alpha, at least for object-oriented code compiled by later versions of **gcc**.

Most often, private symbols are global variables defined by the programmer. However, standard compilers may define global symbols of their own. In some cases this is a well-reasoned decision to optimize the way the data is addressed. For example, MIPS compilers often place constants in the global section where they can be referenced more cheaply using **gp**-relative rather than absolute addresses. Other examples are for implementation convenience at the expense of performance or generality. For example, previous versions of **g++** have defined virtual function tables as writable private data. Even if the tables are not modified, these modules cannot be considered chaste on the MIPS, since **gp**-relative addressing of any kind is prohibited in a chaste module. Moreover, these virtual tables were viewed as writable data of the module that *created* the object, thus no thread can call the object's methods unless (1) it is executing in the same domain as the thread that created the object, and (2) the module that created the object is still attached to that domain, and has not been modified. These restrictions make it impossible to use C++ virtual functions in shared and persistent data, for code generated by the offending compilers.

These compiler choices are reasonably harmless for a private address space system, but not for a single address space system, where the cost of private static data is higher. We encountered one additional unanticipated effect of the single address space on a compiler: some code generated by **gcc** assumes that the GAT is located close to the text section at runtime, which is rarely correct for dirty modules instantiated under Opal. In this case, the compiler codes some case (**switch**) statements using a jump table rather than a sequence of compare-and-branches. To shorten the jump table, the branch target addresses for the cases of the switch are represented not as full 64-bit addresses, but as 32-bit offsets from the global pointer. In a shared address space it is unlikely that a private global block can be located within 4G of the shared code; in any case, there is no way to request this in the Opal model. This problem merely sacrifices an opportunity for a relatively minor reduction in the size of the jump table. However, it can be fixed only by modifying the compiler.

7.3.3 Global Pointer Management for Shared Modules

For fully general handling of private static data, global pointer values for dirty modules must be switched on cross-module calls and returns, by some means other than the PC-relative addressing of the standard Alpha calling conventions. Of course the caller's **gp** can be saved and restored on the stack. The question is how to determine the correct **gp** for the callee.

One approach is for the caller to set the **gp** of the callee before the call is made. The **gp** of statically linked target modules can be saved in the caller's private data, specifically, an array of base pointers mirroring the fanout table proposed in Section 7.2.4. One concern with this approach is that the global pointer values for procedures invoked by runtime linking through a procedure pointer cannot be known; a pointer to a procedure in a dirty module has no meaning unless it is closed by a **gp** value as well.

A second alternative is to retain the current scheme of setting the callee's **gp** immediately after entry to the callee, but to determine the **gp** value by indexing a shared read-only table of global pointer values at fixed addresses near each active module. The table is writable only to the linkage server, which fills in **gp** values as the module is instantiated. Procedures in the module load the module's **gp** from an entry in the table that corresponds to the current context, indexed by a reserved register that holds the current "instance group" identifier (e.g., a protection domain identifier). Disadvantages of this solution are (1) only a fixed number of instance groups can be accommodated, (2) managing instance goup identifiers is tedious, and (3) it consumes an extra register and non-negligible amounts of physical memory.

Perhaps the best alternative is to store the global pointer values for each thread in a table reached at runtime through its thread descriptor. As explained in Section 6.2.1, thread stacks in our Opal prototype are aligned so that a thread can reach its descriptor by masking bits off its stack pointer. The same trick can be used to load global pointer values from the base of the stack with a canned instruction pair. The disadvantages of this scheme are (1) it accommodates only a small fixed number of dirty modules for each "instance group" (as in our current prototype), (2) the table slot for each dirty module must be statically determined (restricting certain combinations of dirty modules), and (3) the global pointer table must be initialized in advance for each thread. All threads in an instance group could share a global pointer array at the expense of an extra memory reference on each cross-module call and return.

Whichever scheme we choose for dirty modules, we cannot avoid copying and relocating the entire GAT at instantiation time, even when it may be only a small number of entries that change. One proposal for reducing the amount of copied data is to divide the LITAs and GATs in two, a *static* part that is immutable, and a *dynamic* part that is relocated at instantiation time. DLLs on OSF/1 are split in a similar fashion so that the statically linked part can be virtually copied using copy-on-write. However, in OSF/1, both parts of the GAT are addressed through the same global pointer. To use this scheme in Opal, each module must have two global pointers. Of course, the static **gp** could be computed PC-relative using the standard Alpha calling convention. However, an additional instruction sequence would be needed for each cross-module call and return to restore the second **gp**, and an extra register is consumed.

7.4 Summary

Opal supports uniform dynamic code sharing through *global static linking*, in which shared symbols are statically linked into the persistent single address space. Global static linking combines the benefits of static and dynamic linking:

- It supports a degree of dynamic code sharing comparable to the most advanced private address space systems using dynamic linking.
- It statically binds the addresses of shared symbols. This reduces runtime overheads for dynamic linking, and produces more efficient call sequences on some architectures, including the MIPS. (However, this benefit may be reduced on the 64-bit Alpha, where all static symbol addressing is indirect unless link-time optimizations are applied.)
- Pointers to all procedures and data items are globally meaningful and can be freely passed and shared.

The single address space makes it possible to combine the benefits of static and dynamic linking in this way. Private address space systems can achieve similar results by globally reserving the address ranges assigned to shared libraries (as in the "Quickstart" facility in OSF/1), moving toward a shared address space. As with the other examples we have given, the price of the flexible sharing made possible by the single address space is that mechanisms are needed to structure and manage the sharing. In this case, Opal needs additional support to allow (1) automatic rebinding to new versions of code modules, and (2) automatic reclaiming of old versions of code modules.

The Opal prototype has achieved uniform procedure sharing, but only for code modules that do not use private static data, for example, pure object-oriented C++ code. For code modules with private static data, the promise of fully uniform sharing -- multiple instances sharing the module's code, but with a private versions of its writable data -- is tainted by additional runtime overheads and restrictions on the number of shared modules that can be attached in each protection domain. We have explored several alternatives for sharing code with private static data by saving and restoring base pointers, but there is no perfect solution. In short, the single address space offers flexible sharing benefits for some programming styles that we consider to be reasonable, even preferable (i.e., object-oriented programming), but it imposes some restrictions and load-time costs if global variables are used frequently.

Chapter 8

Design of an Opal Cluster

Previous chapters have focused on use of a single address space for uniform sharing of code and data within the memory of a single node. This chapter proposes to broaden the single address space concept to support sharing of *persistent* and *distributed* data across a homogeneous cluster of machines with a long-term storage repository. It describes extensions to the Opal model and prototype to support a *network virtual store*, in which segments of virtual memory are preserved on long-term storage and accessed from remote nodes with ordinary **load** and **store** instructions to their assigned addresses in a cluster-wide virtual address space. This is appealing in part because programs can access stored data across the cluster in the same manner as local data, using memory-mapped I/O handled implicitly by the virtual memory system, without converting formats as data moves through the cluster.

The concept of a network virtual store combines elements from many predecessors. Memory-mapped long-term storage originated with Atlas [KELS62] and was fundamental to Multics; it is supported today by file mapping facilities in most modern operating systems. The Bubba project [CFW90] at MCC and preceding work [Tha86] developed the idea of combining mapped storage with persistent virtual address bindings to form a true *single-level store*. Several capability-based architectures, including the Intel 432 and the IBM System/38 and AS/400, support capability addresses in their file systems. The Monads architecture and operating system extended persistent capability-based addressing to a workstation cluster above a coherent paged virtual memory. The creators of Monads view uniform support for cluster-wide sharing and persistence as the primary motivation for a single virtual address space. Certainly the ability to accommodate these extensions naturally and uniformly is one of the model's strengths.

Our goal is to build on the successes of these earlier systems, by extending the Opal model, interface, and implementation to support fully uniform addressing of persistent and distributed data on standard 64-bit processors. Although few of the extensions proposed in this chapter are operational in our Opal prototype at the time of this writing, distribution and persistence concerns dictated a number of fundamental design choices for the Opal system. The single virtual address space is a key aspect of a larger vision to exploit the new potential for memory mapped data access brought by large virtual address spaces, and the tighter coupling of workstations made possible by advancing network technology.

System support for flexible, transparent memory protection is central to the Opal system, and is critical for controlling access to data in a network virtual store. Our design extends Opal's protection and communication facilities so that protection domains can be instantiated automatically for operations on the data they protect. In particular, software capabilities that name user-defined protected objects (Chapter 8) are valid across the network as long as the named objects exist. This is important since these capabilities -- like virtual addresses -- can be passed across the network or stored in long-term data. Protected objects and capabilities are implemented in the runtime system using protection domains and portals; the capability name space is extended to include distributed and persistent objects by means of new kernel facilities for (1) assigning uniform portal names above the persistent network virtual address space, and (2) instantiating protection domains dynamically when their portals are referenced.

This chapter is organized as follows. Section 8.1 explains the motivation for a network virtual address space using an application example and discusses some of the critical issues. Section 8.2 outlines the cluster extensions to the Opal prototype and its kernel interface. Section 8.3 proposes extensions to the protection system and portal facility.

8.1 Motivation and Overview

To motivate a cluster-wide persistent virtual memory, we return to the application example introduced in Chapter 2, an integrated aircraft CAD/CAE system used by the Boeing Company, and under continuing development by Boeing's High Performance Design Systems (HPDS) group. Distributed access to persistent data structures is fundamental to this system: CAD artifacts are saved on long-term storage, then retrieved across a network for revisions or other processing by CAD tools on engineering workstations. The important question is what storage infrastructure best supports collaborative data sharing for applications of this nature.

At the core of Boeing's current aircraft CAD environment is a centralized database system inhabiting a large and growing cluster of IBM 3090 mainframes. The database describes the aircraft parts and their relationships, and indexes tens of thousands of geometry files for manipulation on graphics workstations connected to the IBM cluster. This system has met Boeing's needs for many years, but it increasingly suffers from performance and scalability problems. In particular, it is poorly matched to the needs of the new

CAD tools developed by HPDS, which have three basic properties that were not anticipated by the original database system.

- *Decentralized structure.* Many of the new tools are inherently decentralized: they run close to the display and make intensive use of specialized high-performance graphics hardware. These tools must repeatedly query the database server to fetch needed data structures, typically before they begin executing.
- *User-defined data types and schemas.* The HPDS tools are written in standard programming languages such as C++, whose data formats do not fit naturally in the predefined database schemas. These tools retrieve part records from the database and convert record formats to rebuild their internal structures in local virtual memory on each run. Intermediate results are often discarded and recomputed from scratch.
- *Pointer-based navigation*. The HPDS tools use pointer-based data structures, directly representing relationships among aircraft parts. Pointers allow efficient traversal because they specify the location of the target, but they cannot be represented directly in the database. For example, in a relational database, these links might be represented using part number key fields traversed by associative lookups.

Boeing's HPDS group is exploring alternative database technologies that can efficiently incorporate new programs and information structures as they are developed. New CAD tools must be melded to the database in a scalable way, given anticipated order-of-magnitude growth in the size of the databases and design teams for future projects. Common operations, such as verifying part fits and spatial relationships, must execute quickly to conserve precious design time and reduce costs.

8.1.1 Persistent Stores

The problems facing Boeing are not unique. The causes and solutions for these problems have been debated in the database community since Maier identified the "impedance mismatch" between CAD applications and traditional databases [Mai89]. To better serve the needs of these applications and others, *persistent stores* have evolved that directly support storage and sharing of CAD design information and other pointer-based data structures. With these systems, user programs (e.g., CAD tools) fetch data incrementally from a shared repository on the network and operate on a locally cached image; the system handles I/O automatically as programs navigate through the data by referencing language symbols and traversing pointers. Table 8-1 presents an interface to a simple persistent store.

This approach originated with POMS [ABC+83,CAC84], a persistent runtime system for the language PS-Algol, which was designed primarily to free programmers from the the burden of converting stored data

Table 8-1: A simple runtime interface to a hypothetical persistent store.

Transaction. Begin/Commit/Abort. *Accesses to the store are grouped as transactions to preserve database consistency in the presence of sharing and failures.*

Container.CreateObject (**Type** objectType) returns **ObjectID**. *Create a new object in a designated container, a unit of the persistent store.*

NameService.EnterName (**String** name, **ObjectID** pointer, **Type** objectType). *Register a typed name-to-pointer mapping.*

NameService.Lookup (**String** name, **ObjectID** pointer, **Type** expected). *Retrieve a type-checked pointer to a symbolically named object in the store.*

ObjectID.Dereference () returns **Address**. Fetch the named object from the store if it is not already cached locally; return the virtual address of the cached copy.

structures to and from flat byte-stream file formats. In Boeing's environment, persistent stores allow CAD tools to explicitly represent commonly needed parts relationships (e.g., spatial and functional) in the database. Arbitrary program-generated data structures can be preserved and retrieved easily, encouraging caching and sharing of intermediate results and data views computed by CAD tools.

Persistent stores are rapidly gaining commercial acceptance, with a number of successful products now on the market (e.g., [MS86,LLOW91,Deu91]). Most current commercial offerings are *object-oriented databases* (OODBs) augmented with database features such as query processors, schema definition languages, and indexing facilities. OODBs are distinguished from the earlier *network database model* by extensible user-defined database schemas and type-checked pointers. In many systems, the details of dereferencing object IDs is hidden by the language operator for pointer dereference.

Implementation of these pointers is a fundamental problem for persistent stores. As with other forms of sharing, pointers in a shared persistent store must have a consistent interpretation across all threads accessing the data. On current operating systems with private virtual address spaces, virtual addresses in persistent or distributed data generally lose their meaning when imported into a program's virtual memory, because incoming data is bound late and will not normally occupy the same addresses in each client. Most systems use software-interpreted pointers (*object IDs*) translated to virtual addresses at runtime, typically by hashing or indexing into a mapping table that contains the actual address of the item in virtual memory, if it is resident. Software pointers have been used in distributed programming languages (e.g., Emerald and Orca [BT88]), most persistent stores, and all commercial OODBs. Many of these systems also employ some form of *pointer swizzling*: object IDs are overwritten in place with virtual addresses when brought into memory or registers, to amortize the translation cost over multiple uses of the same pointer.

Pointer representation is tightly intertwined with cache management, which largely determines the performance and scalability of persistent stores. The pointer dereference operator must efficiently locate resident objects in memory, while trapping references to nonresident objects. The system must prefetch and discard data in the cache to reduce access latencies and make the best use of available physical memory. These fetches and discards must synchronize with pointer translation; the system must trap traversals to recently discarded objects, and suspend traversals to in-transit objects until they arrive, to prevent incorrect behavior or duplication of objects in the cache. These needs add to the cost of software pointers, and they can be particulary difficult if swizzling is used.

8.1.2 A Network Virtual Address Space

A single address space operating system such as Opal can simplify many aspects of persistent storage management. As explained in previous chapters, Opal permits direct sharing of cached information among CAD tools on each workstation, which reduces copying and communication overhead. Current tools in Boeing's environment communicate through reads and writes to the shared database, unnecessarily loading the network and server. In most systems, sharing is difficult because tools convert data into process-private pointer structures. Under Opal these structures are easily shared and embedded pointers always have a consistent meaning.

This chapter explores a more ambitious prospect: use of an *extended* single address space as an alternative foundation for naming and cache management in network persistent stores. We propose to extend the Opal system to support a network virtual store, a cluster-wide single address space in which segments of the network virtual memory can be made persistent and recoverable. In such a system, virtual address pointers always have a uniform meaning; each object resides at a unique and stable virtual address, even when it is not in active use by any application. Links in distributed and persistent data can be represented as ordinary virtual addresses interpreted directly by the machine. Cluster members exchange native addresses through messages, and data structures are saved directly on long-term storage and retrieved without the need to translate internal pointers. Database-format object IDs and surrogates are not necessary, although they are of course not prohibited.

A primary benefit of this approach is that storage access is *memory-mapped:* the virtual memory page manager, which controls physical memory allocation and the address mapping hardware, also handles the closely related functions of naming and buffering for persistent and distributed data. Data fetching, caching, pointer trapping, and pointer interpretation are handled in a unified way by the virtual memory system with the aid of MMU traps to (1) detect references to nonresident pages, and (2) synchronize memory references with page fetches and discards. This implies that Opal's strict single virtual-physical mapping is relaxed across the cluster as a whole; that is, a given virtual address may resolve to different physical pages on different nodes. What is important is that (1) each node has a single mapping and (2) addresses have a consistent *interpretation* across the cluster. Within this framework there is a great deal of flexibility for cache management and consistency, as we shall see.

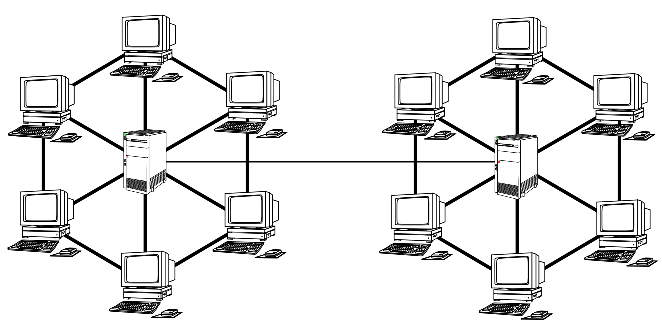
It is widely agreed that memory-mapped database access is preferable in many respects to the traditional separation of buffer management functions into a user-level manager operating within a private virtual memory. This two-level caching structure leads to well-known problems with duplication of function and redundant effort, including double paging, unnecessary data copying, and false dirtying of pages containing clean buffers. Many of these problems are avoided in commercial database systems by statically reserving physical memory for the database buffers. However, this approach is not easily extended to CAD database systems, in which multiple tools are active simultaneously, each with its own database cache. Only the central page manager can determine globally fair and efficient tradeoffs among competing uses of physical memory on each node. The inability to control these tradeoffs sensibly is a primary barrier to scale in current CAD database systems using two-level buffering.

Use of mapped memory, however, relies on operating system mechanisms to support other necessary database functions. Foremost among these is the need for special physical storage management to ensure that all clients see a consistent view of the data. The system must capture updates to local pages, and propagate those updates in a controlled way to nonvolatile backing storage and/or cached copies of those pages in the memories of other nodes. Page faults of shared pages must deliver the correct version of the page. Related needs include transactional synchronization and update, controlled prefetching and discard, aborted updates, integrity checking, and garbage collection. The ability of mapped databases to support these and other necessary functions in a flexible and efficient way is still controversial in some quarters.

We contend that most of the conservative arguments against mapped databases [Sto81], while often cited, have faded over a decade of architectural advances and structural improvements in operating systems. These advances include 64-bit virtual addressing, efficient handling of sparse address spaces, threads and RPC, and extensible virtual memory management. Some technical problems remain, but it will be possible in the near future to construct a complete database environment at user level above a mapped memory substrate provided by a general-purpose operating system kernel. For example, protocols for recoverable and coherent virtual memory have undergone almost ten years of refinement, and a wide variety of protocols now exist, supported by programmable virtual memory interfaces on newer operating system kernels. We defer discussion of specific policies to Section 8.2.6.

Recent advances notwithstanding, it is important to acknowledge the limits and difficulties of a network virtual store for a complex database environment like Boeing's. The data management problems handled by database systems do not simply disappear in a network address space. In particular, uniform addressing alone is not sufficient to preserve the continuity of virtual address pointers in persistent data. A virtual address is a machine-dependent reference to one version of an object at a fixed location in a particular cluster address space; it must be relocated if its target is copied or moved to a different part of the address space, e.g., for compaction or reclustering, or to introduce a new version of an evolving structure. A pointer must also be relocated if it or its target is transferred to a node with different data representations or a different addressing domain (e.g., to a narrow-address machine or a different cluster).

All of these issues must be dealt with in some form by any network persistent store. The purpose of a network address space is not to entirely *eliminate* existing mechanisms to process and convert pointer-based data structures, but rather to reduce the *frequency* with which those mechanisms are applied. In particular, the network address space can eliminate the need for conversion for the most common transfers: between programs on a single node, between memory and long-term storage, and between homogeneous nodes within the cluster. This can be viewed as "caching" an efficient machine-dependent representation of the data, while retaining the ability to convert to a machine-independent format in exceptional cases, e.g., transmission outside of the local addressing domain.



Two homogeneous Opal clusters connected by a network. Inter-cluster data sharing requires by-value communication of linearized or "swizzled" (relocated) data. Intra-cluster sharing uses distributed shared memory with cooperative caching of cluster data in the memories of the cluster nodes.

Figure 8-1: Sharing and cooperation in networked cluster virtual memories.

Opal was designed as a foundation for such a mapped network repository. Mapped persistent and distributed data are special cases of shared memory, the basic focus of this dissertation. The mechanisms and issues discussed in previous chapters for code binding, symbolic naming, synchronization, and protection of shared data all apply directly in the context of a network virtual store. Within each node, pure protection domains can be used to prevent cooperating database tools from viewing or modifying parts of the database not directly required for their function, without impeding sharing of the cached state when access permissions naturally overlap, e.g., the output of one CAD tool is the input of another. Across the cluster, the uniform address space allows database cache managment to be viewed as a distributed shared memory problem, to which techniques developed for high-performance distributed-parallel programming can be applied, including synchronized client-to-client data transfers and relaxed consistency [GLL+90].

In summary, we believe that architectural support for large, sparse virtual memories is the catalyst for wider use of memory-mapped storage access, which will ultimately offer the best performance and scalability for integrated information-intensive applications like Boeing's collaborative CAD system. Various operating system advances have combined to make this approach viable on a large scale. Opal and the extensions proposed in this chapter build on these advances with (1) a network-wide, persistent virtual address space, (2) flexible primitives for protecting data in the virtual memory store, and (3) system hooks for configurable data management within the global virtual memory. These features support persistent and distributed pointer structures in a natural, uniform, and efficient way, replacing the software pointer techniques commonly used above private address space systems on 32-bit architectures.

8.2 Extending Opal for a Network Virtual Store

This section describes additions to the prototype structure outlined in Chapter 6, to extend the Opal system facilities across a cluster with a persistent network virtual address space. The network virtual store is a conceptually simple extension to the Opal model already defined:

- Opal segments can be made logically persistent by means of persistent reference counts, as described in Section 4.4. Objects allocated from memory pools in persistent segments continue to exist at their assigned virtual addresses. In essence, memory pools are equivalent to the "containers" in Table 8-1.
- Storage and address space for persistent segments is accounted by resource groups, which persist as long as they contain persistent resources or until they are explicitly destroyed.
- The system must partition the virtual address space across the cluster so that each virtual address has a unique meaning across the cluster, and so that segments can be located using the *AttachByAddress* interface. Symbolic name spaces must also be distributed in some fashion.
- Capabilities that name the Opal system objects (segments, protection domains, and resource groups) must be valid from anywhere in the cluster. Also, capabilities for persistent objects must remain valid as long as those objects exist.

This section explains our implementation approach and explores some tradeoffs and unresolved problems. Two basic design considerations should be noted in advance. First, cluster members act autonomously whenever possible, so that the network will generally have no performance impact unless data is shared across the network at a fine grain. Second, client workstations in the cluster are trusting only to the extent that user code directs them to share resources. Damage from a failed or corrupt node is limited to threads that depend (transitively) on resources shared with that node. The design assumes that network communication is authenticated and secure, as explained below.

Section 8.2.1 is an overview of the cluster architecture and how nodes communicate and share data. Section 8.2.2 describes how the network address space is partitioned across the nodes in the cluster, and Section 8.2.3 explains how each node manages its data, and accesses data allocated on other nodes. Section 8.2.4 outlines the handling of capability-based operations on Opal system objects. Section 8.2.5 and Section 8.2.6 discuss some issues for managing recoverable objects and segments in the cluster.

8.2.1 Cluster Structure and Inter-Node Communication

An Opal cluster consists of a trusted *cluster server* and a collection of untrusted client nodes. Each node runs an instance of the Opal server described in Section 6.1, which is viewed as an *agent* of a logically distributed Opal service. In our current prototype, the cluster server is an Alpha OSF/1 system running a Network File System (NFS) server and a modified version of the Opal server (the *master agent)*. The cluster server coordinates the actions of the clients and provides all backing storage for the global virtual memory, accessed by client nodes through the NFS paging server described in Section 6.1.2.

Peer agents cooperate directly to manage sharing and maintain a uniform space of capabilities for Opal system objects. For example, peer agents can share segments in part by exchanging NFS file handle tokens so they may page data from shared backing files on the cluster server. Communication among the agents is currently based on Open Network Computing (ONC) RPC, a public domain RPC facility developed by Sun Microsystems for NFS and other Unix-based distributed services. ONC RPC bindings can be passed by value.

To arrange RPC connections among client agents, each cluster node is uniquely identified by a *node number*; any agent may obtain an RPC binding for any peer agent by node number. The master agent maintains a table of node numbers and RPC bindings for all active agents; other agents keep local caches of the central node tables. Each agent receives its node number at boot time, along with the network adress of the cluster server, which can be used to obtain a binding for the master agent from an ONC binding clerk. The booting agent registers with the master agent, passing its node number and a binding for itself. Updates to the central node tables propagate lazily to the rest of the network; each node service peer agent lazily accumulates RPC bindings for the nodes it interacts with. These cached tables may become stale as nodes arrive and depart asynchonously without global notification. Stale RPC bindings in the cache are detected on use and refreshed by retrieving a new binding from the master agent.

8.2.2 Managing the Network Address Space

Partitioning of the network address space is hierarchical. The master agent allocates large fixed-size *chunks* of address space on request from client nodes. Each node subdivides its chunks to satisfy segment allocation requests from local threads. Thus most segments are allocated locally without contacting the cluster server, and the segments allocated throughout the network are assigned disjoint address ranges. Chunks are small enough to ensure that there are many more chunks than nodes, to handle unbalanced allocation if nodes consume address space at different rates; however, the maximum segment size is limited by the chunk size. A full 64-bit virtual address space could provide, for example, one million 16-terabyte chunks, perhaps sufficient for a network of a few thousand nodes.

The master agent maintains a complete record (the *chunk table*) of allocated chunks and the node numbers of their owners. Clients use this information to locate segments for *AttachByAddress*. Remotely owned segments are located by querying the cluster server for the owner of the containing chunk, and forwarding the request to the owner. Chunk ownership requests are most often satisfied from a local client cache of the chunk table, acquired lazily in a manner similar to the node cache of RPC bindings. Stale chunk mappings can occur only when chunks are returned to the cluster server, but in any case they can be detected on attempt to attach a segment within the chunk, trusting the previous owner to report that the entry is invalid.

Table 8-2: Basic client-to-master RPC interface for clustered Opal systems.

ClusterParams: {**Address** start, **ByteCount** chunksize, **ByteCount** rootsize}. *Address space parameters fixed for the lifetime of the cluster .*

ChunkInfo: {**Address** start, **NFSHandle** dir, **NFSHandle** root}. *Chunk descriptor, conferring ownership of a chunk and its backing file pool directory.*

CheckinNode (**NodeID**, **RPCHandle**) returns (**ClusterParams**, **ChunkInfo**). *Join a cluster as an agent with an authenticated node number,. Receive cluster parameters and an initial chunk, including agent's root segment.*

ResolveNodeID (**NodeID**) returns **RPCHandle**. *Request an RPC binding for a peer agent, specifying its node number.*

AllocateChunk () returns **ChunkInfo**. *Allocate a fresh chunk of address space to be owned and managed by the calling agent.*

ChunkOwner (**Address**) returns **NodeID**. *Determine the owning node of the chunk containing the specified virtual address.*

Note that larger networks can be accommodated by extending the allocation hierarchy. This changes our trust rules somewhat: each agent must trust all servers on its path up to the root of the cluster, and down the branch to any other agent whose resources it uses. One weakness of hierarchical address space allocation is that it is difficult to reclaim address space at the chunk level, since any chunk is likely to contain allocated segments at any time.

8.2.3 Network Segment Pools

Each chunk of address space is managed by its owning node as a separate self-contained *segment pool* (Section 6.1.1), backed by a separate directory of backing files (a *file pool*) on the cluster server. Segment pools are self-described by a *root segment* occupying a fixed-size block of address space at the base of the pool's chunk. The root segment is a persistent heap containing all metadata structures pertaining to the pool, including address space allocation and mapping tables, *Segment* and *SegmentRef* objects, and access control lists for published segments. When the master agent assigns a chunk to a client node, it creates the file pool directory and a root backing file within that directory, and passes the NFS file handle tokens for the directory and root file to the owning client agent. The agent attaches the root segment backing file at its assigned address, initializes the root segment in its virtual memory, and uses *MemoryPool* primitives internally to allocate chunk metadata from the root segment. The segment pool and backing file directory are managed and paged as described in Section 6.1.1 and Section 6.1.2.

Because each chunk is self-contained, chunk ownership can transfer to another node if the original owner fails or retires, so that its persistent resources can continue to be served on the network. This *fail over* involves updating the central chunk table and passing the pool's root segment token to the new owner. The single virtual address space guarantees that the metadata for any segment pool can be attached and interpreted by any agent. However, no agent can access another agent's pools or segments unless the owning agent or cluster server grants it the required NFS tokens. Note also that all metadata sharing is sequential and validated before use; agents are mutually distrustful and never share data concurrently.

Within each client agent, the chunk table holds pointers to the *SegmentPool* objects for all chunks owned by that node. Local *AttachByAddress* and *NewSegment* requests are satisfied by first determining the containing *SegmentPool* (e.g., by indexing the chunk table), then delegating the request to that pool. These operations typically complete without network activity.

Each agent maintains a private local *surrogate* segment pool for each remotely owned chunk that contains segments attached locally. A surrogate pool is a *SegmentPool* object with the same interface, procedures, and data structures as an ordinary segment pool. Like an ordinary *SegmentPool*, a surrogate pool is

Table 8-3: Basic peer-to-peer RPC protocol for clustered Opal systems.

SegInfo: {**Address** start, **ByteCount** len, **NFSHandle** file}. *Segment descriptor, allowing the receiving agent to attach the segment and page it locally.*

SegRefInfo: {**SegInfo** seg, **SegProt** perms, **SegmentRef** ref}. *Descriptor for a SegmentRef, specifying permissions and capability.*

ResolveAddress (**Address** gva) returns **SegRefInfo**. *Look up a published segment by probing address mapping structures with a contained address.*

SegmentRef.Reference (). *Acquire a reference for a remote* ***SegmentRef****.*

SegmentRef.Release (). *Notify an agent that all local reference counts on a remote* ***SegmentRef*** *have been released.*

ResourceGroup.Reference (). *Notify that a local resource is charged to a remote resource group: validate the group, and register the calling agent for a callback (DestroyGroupNotify) if the group is destroyed.*

ResourceGroup.Release (). *Notify that all local resources in the remote group have been released, releasing the request for a* ***DestroyGroupNotify*** *callback.*

ResourceGroup.Destroy (**ResourceGroup** parent). *Destroy a remote resource group, presenting a capability for the group and its parent.*

DestroyGroupNotify (**ResourceGroup** rgc). *Notify a remote agent that a locally owned group in which it had a registered interest is being destroyed.*

indexed by the local chunk table entry for its chunk. The surrogate pool contains information about the chunk and its segments, obtained from the master agent and the chunk's owner, as depicted in Figure 8-2. The surrogate pool's address map and segment table contain private cached records for locally attached segments from the remote chunk, containing the segment's address range and NFS backing file token. Note that surrogate pools never access the chunk's metadata. For example, new segments cannot be created in a surrogate pool; only the chunk's owning node can allocate segments from the chunk.

8.2.4 Handling Remote Opal Resources

In our current prototype, remotely owned segments may be attached only by symbolic name or by virtual address, using variants of *LookupName* and *AttachByAddress* described in Chapter 4. Our prototype uses a central global name server built into the master agent; a *LookupName* request deflects to the master agent if the requested name is not defined locally.

To generalize the full Opal interface to operations on remote objects, capabilities must have a consistent interpretation across the cluster. This is needed, for example, to attach a remote segment by capability, or to charge a newly created segment to a remote resource group. The scheme for interpreting capabilities for remote objects has three basic elements.

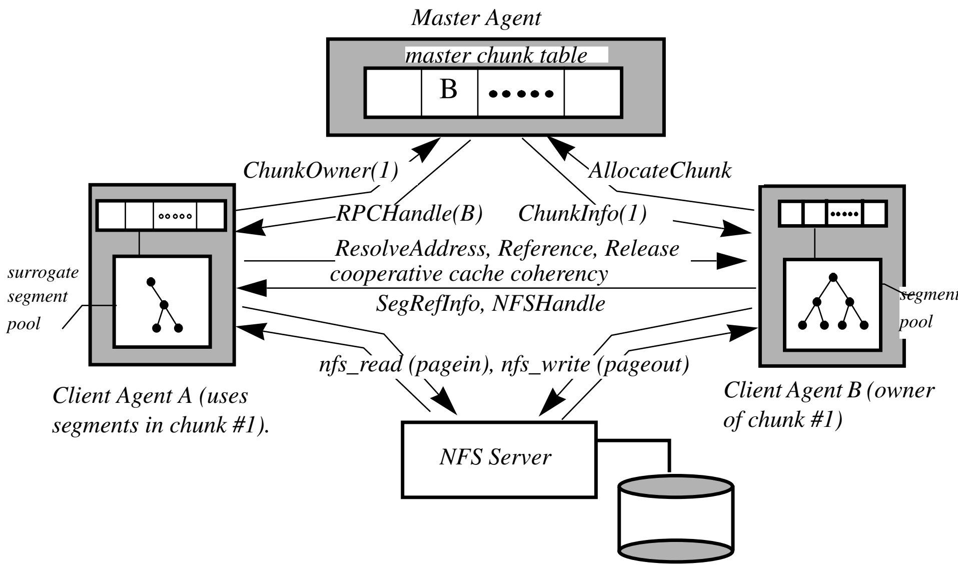


Figure 8-2: Managing a shared segment pool in a network address space.

- All operations on any Opal object are initially received by the agent on the node where the call originated. Every Opal agent serves the same statically determined portal IDs, thus a capability for any Opal object in the cluster always binds to the local agent.
- The receiving agent determines if each argument for an incoming call is a local or remote object. An object's *home node* can be determined from its capability: it is the owner of the chunk containing the object's guard, determined from the guard address in the capability.
- Some operations on remote objects are handled from a local *surrogate* for the object. For example, most segment operations are handled from cached state in a surrogate segment pool, as described above. Most resource group operations are also handled locally. Table 8-3 presents a partial peer-peer interface for these objects. Other operations (e.g., attaches to remote protection domains) are simply forwarded to the object's home node.

The challenge of this scheme is that each agent must trap incoming calls on remote objects in a way that does not impact performance for operations on local objects. The chunk table must be indexed for each incoming capability to determine if it is local or remote. If the object has a locally cached surrogate, it is located by hashing. Section 8.3.2 discusses extending this scheme to user-defined services.

Remote resource groups illustrate how surrogate objects can be used. Each agent keeps a local surrogate for each remote resource group with local members. If a local resource is charged to a remote group, the local agent registers an interest in the group with the group's home node using the *Reference* function in Table 8-3. Additional local objects can be charged to the group without contacting its home node. When the last local resource is removed from the cached group, the surrogate is discarded and the home node is notified with *Release*. Requests to destroy a resource group are forwarded directly to the owning agent, which validates the request and notifies all interested agents of the event with *DestroyGroupNotify*. Each agent can always release its own resources, regardless of their group. However, no agent can manipulate a remote group unless the needed capabilities are explicitly granted to that agent. For example, no agent can destroy a remote group unless it possesses capabilities for both the group and its parent, both of which must have been passed to it by user code.

8.2.5 Persistence of Opal Resources

Distribution is closely tied to recoverability, since nodes can fail independently. For example, any Opal resource that is referenced by another node must be retained even if its owning node fails and recovers. If the holder of the reference fails, it must not "lose" the reference. In addition, each agent must retain objects made persistent through the resource control primitives.

Section 6.1.6 explained how an agent recovers its internal state and lazily rebuilds the underlying microkernel state after a shutdown. In the cluster, the agent's root segment and epoch are provided by the master agent when the restarting node registers with the cluster server. The cluster server preserves chunks and backing files assigned to a downed node, thus all persistent state created by the node can be recovered at the assigned addresses. Some internal structures (e.g., the chunk table and node table) are discarded and rebuilt lazily on the clients. Capabilities for Opal resources are stable across restarts, because every agent serves on statically defined portal IDs, and validates capabilities through guards recovered at stable virtual addresses.

This outline for handling persistent Opal resources assumes that metadata updates are propagated to backing files in a controlled way, so as to leave them in a consistent state. Our current prototype handles only planned shutdowns, in which modified pages are flushed to their backing files. As explained in Section 8.2.6, we plan for each agent to commit records for operations on persistent internal state, to be scanned and reapplied on recovery.

Agents of an Opal cluster cannot always recover from failures independently of other nodes in the network. Although most agent state is private, node recovery must be coordinated if cross-node resource references are held. For example, agents may charge local resources to each other's resource groups, or hold local references on remote *SegmentRefs*. These cases are equivalent, so we consider the case of a local resource charged to a remote group. In this case, both the *holder* of the reference and the *owner* of the referenced group must keep a recoverable record of the reference. The owner's record is committed first; if the owner fails, it will recognize the reference on recovery, and retain the group even if it has no local members. However, there are several difficulties if the *holder* fails.

- If the holder fails before committing its record, then the resource may persist incorrectly until the group is destroyed. This is inelegant, but the holder can prevent it at some cost by committing an intention before registering the reference, and checking recently committed intentions on recovery.
- The failed node's resource references will never be released until it recovers. We assume that all failed agents eventually recover, or that the resources of a permanently failed agent are failed over to another agent (see Section 8.2.3).
- If the group is destroyed while the failed holder is unreachable, the *DestroyGroupNotify* message must be held (e.g., by the master) and passed to the agent when it recovers.

All non-trivial distributed services face similar problems. Alternative solutions include: (1) streamline the failure guarantees to allow a "stateless" protocol such as NFS, (2) rebuild some persistent state of a recovered node by calls from its peers [Mog94], and (3) use a fully general two-phase commit protocol. In this case, the distributed state must be retained across failures of both the holder *and* the owner, so the first two alternatives are inadequate, but the operations are simple enough that general two-phase commit is unnecessary.

8.2.6 Coherency and Recoverability

This section has outlined the implementation of an Opal cluster, on the assumption that segments of the global virtual memory -- including metadata segments -- can be made recoverable and/or coherent. Our current prototype uses limited physical storage management policies for consistency: persistent segment pages are flushed to the backing file on last detach, and all segments are flushed on shutdown. This policy is sufficient for sequential segment sharing across the cluster, and persistence in the absence of failures, but more general solutions are needed.

As stated in earlier chapters, most aspects of data management in Opal are under the control of user code; many data models and data management policies are possible, and the operating system should not dictate a single solution. Policies for consistent storage management should be selected on a segment grain, in this case provided by *cache managers* involving some combination of support from the compiler, runtime system, and custom paging servers using kernel interfaces for application-controlled memory management. For example, the external memory management interfaces in first-generation microkernels such as

Mach [You89] and Chorus [ARS89] permits user control over page protection and fault handling, which can transparently support a range of consistency protocols [Duc89, SZ90, LH89] and other data management services surveyed in [AL91]. More recent extensible operating systems have refined these interfaces further [HK93]; some interfaces also allow control over page cache residency to improve performance [HC92,LCC94]. Programmable kernels are currently a focus of intensive research, and we anticipate a continuing trend toward more flexible and efficient facilities for application control over memory management.

Modern high-performance protocols for recoverable and coherent memory often rely on directives from the compiler, runtime system, or user code. For example, fine-grained or "multiple-writer" protocols [e.g., KCZ92] with relaxed consistency models [GLL+90] are driven by synchronization primitives. Similarly, recoverability may be based on transaction or checkpoint primitives, perhaps augmented with cache control directives (e.g., prefetch and discard). Some consistency prototcols rely on marking of modified byte ranges by user code, or object-grained coherency as we have advocated with Amber [CAL+89]. These approaches are compatible with Opal's network virtual address space; indeed, we believe that they are required for good performance. The consistency guarantees for each segment are defined by a "contract" between its cache manager and the user programs accessing that segment. A user program that does not observe the conventions required by the contract (e.g., proper use of synchronization or commit primitives) may experience poor performance or an inconsistent view of the segment. Different contracts may be in effect for different segments in the global address space, reflecting a continuum of consistency solutions balancing transparency and performance.

Physical storage management concerns are the primary focus of continuing research in the Opal group. This research has produced several papers and prototypes, but the mechanisms have not been integrated into our Opal prototype. For example, we are building a transactional cache manager intended for the collaborative design environments discussed earlier. In this system, transactions are threads operating on virtual memory with user-level synchronization. Each transaction executes within a single node, updating locally cached pages from one or more segments. Modified byte ranges are written to a log segment on commit; log records are scanned and reapplied on recovery. Log records are propagated to peer nodes with cached copies of modified segment data, to keep these caches consistent. We described a prototype of this transactional *log-based coherency* approach in [FCNL94].

The transactional cache manager prototype is based on the logging and recovery package described in [SMK+93], which is incompatible with memory-mapped segments. Each segment is read into transient virtual memory when it is attached, and is paged to a local swap area to prevent pageouts from overwriting the backing file with uncommitted data. To support fine-grained transaction commit in mapped memory, we intend to *shadow* backing files at the file server. Shadowing can support recoverable segments without logging, but in our system their purpose is to accept pageouts without overwriting the backing file until a *checkpoint* operation issued by the transaction package, which atomically updates the backing file block map. A similar scheme was considered in [Tra82]. We are prototyping shadowed files as an extension to the log-structured file system implementation described in [SBMS93].

The combination of user-level logging and shadowed backing files has a number of advantages. Fine-grained transactions are efficient because they log only the data that was actually modified, and transaction support is kept out of the lower levels of the system (e.g., the microkernel and file server). We intend to use a variant of this approach to recover the internal state of the Opal agents themselves, by logging *logical* records, rather than new object values, for agent operations on persistent Opal system objects. There is no need to maintain coherency for these agent metadata segments, since they are never shared concurrently. For user segments shared across the cluster, log-based coherency exploits the logging mechanism used for recovery to preserve transaction consistency in an efficient and flexible way. This approach to recoverability and coherency does not require a special paging server, but it relies on user code to identify modified objects and to synchronize data accesses correctly.

These techniques are the elements of a cluster-wide virtual store for a tightly-interacting cluster of workstations. The single address space simplifies use of client-client data transfers -- characteristic of distributed shared memory systems for parallel programming -- to improve performance for collaborative applications like the Boeing CAD system, in a way that exploits today's large memories and fast networks. The memories of workstations in the cluster can be viewed as a global cache over the repository, with the goal of satisfying most paging operations with memory-to-memory network transfers rather than from disk, as described in [FMP+95]. With the transactional virtual memory model outlined above, these *cooperative caching* techniques [DWA94] can be applied in a fault-tolerant way, since committed updates can always be reconstructed from the log. However, fine-grained coherency protocols, including log-based coherency, complicate cooperative paging because multiple versions of each page may exist in the network. These issues are the subject of current research.

8.2.7 Discussion

This section outlined a scheme for serving Opal resources in a network-transparent fashion, and a longer-term vision for a workstation cluster operating system that both supports and exploits the single address space. Cooperating agents on each node interact under the direction of a central server to extend the Opal system facilities across the cluster. In this cluster architecture, most system operations complete locally as described in Chapter 6; agents interact only to handle operations on shared resources. Physical storage managment for consistency, residency control, and cooperative caching are provided by configurable segment cache managers.

Several points are worthy of emphasis:

- The central server in our prototype limits scalability and is a single point of failure, but the conceptual structure is sound. The file servers and address space servers can be replicated for reliability and availability, and larger clusters can be constructed by extending the address space partitioning hierarchy upwards, and/or adding file servers.
- The master agent is trusted, but client agents are mutually trusting only to the extent that user threads on those nodes direct them to share resources. All resource access is controlled by Opal password capabilities and NFS file handles, which cannot be illegally obtained or forged. However, these security claims rest on certain assumptions about network communication: (1) packets containing capabilities cannot be stolen, (2) the source of every message is authenticated by node number, and (3) messages cannot be intercepted or modified by a malicious third party. These needs can be met by existing protocols for authentication and encryption.
- Although the Opal server agents maintain recoverable state, only a few operations involving remote resource references actually require coordinated updates on two nodes. The restricted nature of these operations can be exploited to keep the recovery protocol simple.

8.3 Protection in an Opal Cluster

The distributed Opal system outlined in the previous section is a distributed service composed of cooperating persistent agents. Capabilities for the protected objects that comprise the service (segments, domains, and resource groups) can be interpreted consistently by any agent, and are valid across restarts. This section proposes extending Opal's protection and communication facilities to support *user-defined* services built above the network virtual store. We consider object-oriented services that use the protected object runtime package presented in Chapter 5. The goal is to preserve the validity of capabilities for protected objects across the cluster and across restarts, so that -- like virtual address pointers for an object -- they are valid as long as the target object exists. The concepts outlined in this section could be applied to other services that are not object-oriented.

The term *service* refers to any component invoked across a protection boundary. Our general goal is to preserve protection structures for data that is persistent or shared across the cluster. The isolation of protection domains is essential for a network virtual store to be used safely. In a complex application system like Boeing's CAD environment, multiple protection domains with different privileges will exist at runtime, perhaps protecting different CAD tools running on behalf of different designers, as described in earlier chapters. Threads in these domains cooperate through overlapping access to segments of the network virtual memory, and by cross-domain RPC calls through portals. The domains and threads are initially created by applications using Opal system call, but it would be tedious for application code to reconstruct them after each restart, along with the sharing and RPC connections among them. Instead, we propose to view protection domains as logical attributes of the data they protected, instantiated automatically when that data is used, i.e., by invoking a protected object.

A number of systems, including Clouds, Eden, and Arjuna [DPSW89], have included some notion of long-lived protected objects accessed across a network through uniform software capabilities. In Opal, distribution and persistence for protected objects is supported at the run-time level, and derives directly from distribution and persistence of the single address space. A service is invoked through a portal that is named by an integer portal ID embedded in the capabilities for its protected objects. The key idea is that each portal ID value represents a *logical* service whose state and threads are encapsulated by a protection domain that is instantiated automatically when the portal is used. Segments that contain the service's code and internal state must be attached to the new domain, and some service-specific recovery or initialization may be required. Once this is complete, capabilities for protected objects served through the service's portals will bind to the new domain, which can interpret them.

Section 8.3.1 explains persistent protection domains that reclaim their portal IDs on recovery, so that capabilities issued by the service are stable across restarts. Section 8.3.2 extends this idea to distributed services composed of cooperating agent domains serving a common portal ID on multiple nodes.

8.3.1 Persistent Protection Domains

To support persistent services, we propose a new system call to mark a given protection domain as **persistent** (see Table 8-4). The caller passes a capability for a *ServiceControl* object defined by the service, which assumes responsibility for any service-specific initialization and recovery. If the system restarts, any portals assigned to the persistent service will be reserved. If any thread attempts to switch through a reserved portal, the domain is reinstantiated, and the *ServiceControl* recovery procedure is invoked.

The Opal server preserves all state needed to rebuild a persistent protection domain, including its portal set, its segment attach list, and the *ServiceControl* capability. To reinstantiate the domain, the server creates a new domain with a freshly initialized standard runtime system, reattaches the segments in the domain's attach list, reregisters its portal set, and upcalls the user recovery procedure. The refurbished domain can then accept calls through its portals.

Recovery of the service's internal state is the responsibility of its segment cache managers and the recovery procedure. Any segment recovery protocol is completed by the segment's cache manager before that segment is used. The service recovery procedure must then restore other domain-specific state, particularly state residing in the primary data segment, which is transient. For example, thread objects are preserved if they are allocated from persistent memory, but they must be replaced on the scheduler's ready list, which resides in the primary data segment. In addition, static data items (if any) will be reset to their initial values. Thus the service must maintain any state needed for recovery in data structures allocated from persistent memory, and rooted in the object passed as an argument to the recovery procedure. This is similar to the recovery procedure of an Opal agent, which is one example of a recoverable server.

Clients rebind to a recovered server automatically when the portal ID is reassigned. However, the RPC protocol built above portals must detect that the portal was rebound, and reinitialize any channel state. For example, in our current prototype all portal communication is through Mach ports, whose rights may be indetectably stale if the node has failed and recovered. The client can detect a stale right by comparing an epoch number in the channel with the current epoch, and rebind to the service (with *ResolvePortal*) if the epochs do not match.

8.3.2 Distributed Services and Service Agents

A similar portal trap scheme can permit distributed access to services by portal ID, and permit uniform network access to user-defined protected objects. The simplest way to ensure a consistent interpretation of portals and capabilities across the cluster is to assign portal IDs uniquely across all the nodes, and resolve a remote portal ID by locating the server domain and returning a network RPC binding for it. (As noted in Section 6.1.4, portals are allocated from the virtual address space, thus they are guaranteed unique across the cluster.) However, with this scheme, all services are centralized and fixed on a particular node.

We propose instead to extend the agent-based scheme used for the Opal service, and handle calls on remote portals by instantiating a *local* protection domain to act as an agent of the service, serving the same portal IDs -- and therefore the same capabilities -- as the remote domain. The aggregate of agents for a given portal is in essence a *distributed* protection domain defining a *logical* service accessed through a particular portal ID. For instance, suppose a thread on node **B** attempts to invoke an object created by an agent of a distributed service on node **A**, which indirectly passed to **B** a capability for the object, containing a portal ID for the service. The capability was minted on **A**, but the agent on **B** serves at the same portal ID, which always resolves to a local agent if one exists. The agents cooperate to maintain global consistency of the internal service state.

Table 8-4: Agent and service control interface.

Domain.MarkPersistent (**ServiceControl** control, **Address** argument) *marks a protection domain as persistent. The arguments are used to notify user code if the domain is reinstantiated.*

Domain.InstallAgent (**NodeID** node, **ServiceControl** control). *System call request to lazily create a peer agent for the specified domain on the specified node.*

ServiceControl.PrepareAgent (**NodeID** node, **Domain** agent). *System upcall to an active service agent, a request to prepare a remote domain to act as a peer agent representing the service*.

ServiceControl.Startup (**Address** argument). *Agent-to-agent call to initialize a newly created service agent.*

ServiceControl.Recover (**Address** argument). *System upcall to a reinstantiate a service agent, a request to initiate application-specific recovery of the agent state.*

This structuring of services as a group of cooperating agents is a useful design choice because the agents may keep local copies of the data maintained by the service, for reasons of reliability (replicas), performance (caches), or availability ("stashes"). For example, local agents can reduce network communication by caching a service's protected objects and executing some methods on the cached copy without involving remote agents. Agent-structured services may be a useful technique to exploit service locality in the next generation of switched-mesh interconnects. Experience with coherent page cache managers in Ivy [LH89] and Mach [FBS89] have measured performance benefits for this structure by reducing communication on older Ethernet networks.

The structuring of distributed services implies that capabilities, like virtual addresses, may resolve differently according to the node where they are used. This distinction between logical service and physical instance, supported by the naming system, permits the service implementor to decide the degree of centralization or distribution to use, as well as to change that decision dynamically. The distributed nature of the service is encapsulated within the service itself. Clients use the same code to access a local service or a distributed service; agent binding is transparent to the client. A protection boundary isolates the cached service data so colocated clients can safely call it; the cached data could be shared readonly by the clients, but the agent controls all updates.

The *InstallAgent* primitive (Table 8-4) creates a peer agent for a given domain on a specified target node of an Opal cluster. Suppose *InstallAgent* is invoked to create a peer agent on target node **B** for a protection domain on node **A**. The Opal server agent on **A** sends a message to its peer on **B**, passing the target domain's portal set, a capability for the resource group containing the target domain, and a capability for the *ServiceControl* object.

If a thread on **B** calls through a portal in the portal set, the Opal server on **B** instantiates the agent by creating a fresh domain charged to the specified resource group. It registers the portal set for the new agent domain and upcalls *PrepareAgent*, passing a capability for the new domain as an argument. *PrepareAgent* initializes the agent by attaching segments and executing code of its choosing within the new domain, e.g., the *Start* operation in Table 8-4. When the new agent is ready, *PrepareAgent* returns to the Opal server on **B**, which then permits threads to switch through the new agent's portals.

This assumes that it is possible to issue capabilities (e.g., for a *ServiceControl*) that bind to a particular agent, rather than the closest one. Opal capabilities contain a reserved nodeID field so that the caller may request a binding to a particular *instance* of a portal, and invoke its agent with network RPC calls. However, our prototype does not support RPC calls to remote portals.

8.3.3 Discussion

This section proposed extensions to Opal's protection and communication facilities to allow applications to construct persistent and distributed services. Clients access the service through a portal that is valid for the lifetime of the service, independent of location. Preserving the continuity of portal IDs endows password capabilities with the same degree of uniform referencing as virtual addresses. Recoverability of protected objects and persistence of capabilities derives directly from the persistent virtual memory. Opal is itself an example of such a service, accessed through statically defined portals that are known to all Opal programs.

These ideas depend on and build upon the network virtual memory. Although these facilities for persistent and distributed services are largely independent of the single address space concept, Opal's passive protection domains and named portals lend themselves to dynamic instantiation, and the network address space defines a uniform name space for the service state, both internally with virtual addresses, and externally with password capabilities derived from virtual addresses. Opal applications can exploit this uniformity using the proposed extensions to control the lifetime and placement of protection domains associated with particular portals.

8.4 Summary

This chapter outlined an approach to extending the Opal system to a cluster of independent 64-bit nodes and a long-term storage respository. We proposed a broadening of the single virtual address space to a *network virtual store*, in which segments of a single cluster-wide virtual address space can be retained in long-term storage and accessed through uniform virtual addresses from any node in the cluster. Boeing's collaborative CAD system is an example of an application environment that uses persistent and distributed data structures, and that benefit from a network virtual store. The primary points of this chapter are:

- Architectural support for sparse 64-bit addressing facilitates the use of *memory mapped* storage access, which offers superior performance, scalability, and generality for applications using persistent data. A primary goal of the Opal project is to provide operating system support for general and flexible use of these techniques. In particular, Opal accommodates multiple approaches to physical storage management for coherency and recoverability, using application-specific *cache managers* that control propagation of updates to backing store and to peer memories.

- In a network address space, data can move through the cluster without converting pointers or data formats. Virtual addresses can be exchanged on the network or through shared persistent storage, facilitating storage and sharing of complex data structures. Nodes using common data in the network virtual store can receive data directly from the memory of a peer, which is often faster to access than the server's disk on next-generation networks. However, user code must employ existing mechanisms to convert formats for data moved *outside* of the cluster.

- A persistent network address space system can be constructed using standard microkernels and file servers. The elements of our approach are: (1) system services are provided by cooperating system server *agents* on each node; (2) address space is allocated hierarchically among the agents by a trusted service; (3) agents preserve their internal state in virtual storage, and reconstruct microkernel state as needed to support persistent system objects; and (4) agents share resources by exchanging unforgeable access tokens (capabilities), e.g., NFS handles for backing files of shared segments. In our current architecture, client agents are trusting only to the extent that user code directs them to access specific shared logical resources.

- Pure protection domains restrict each component's access to the global mapped store in a flexible way. If coresident components have common access to segments of the store, the single address space allows them to share a memory copy of any cached pages (using sharing facilities described in previous chapters) for the best performance and memory utilization. The cluster design includes extensions for persistent and distributed protection domains, so that capabilities for user protected objects are valid across the network as long as those objects persist. The Opal system itself is both an example of a persistent and distributed service, and the provider of facilities needed by user-defined services.

- Managing distribution requires attention to the performance implications of the network, involving caching and a weakening of the consistency guarantees provided on a single node. Although names have a uniform *meaning* throughout the cluster, at a lower level the *binding* of these names to physical data may vary to accommodate caching, replication, and migration. Consistency of the named objects is defined by a “contract” between the applications and the service supporting the distributed resource. This principle affects Opal’s handling of both data names (virtual addresses) and resource names (capabilities).

The proposed architecture, grounded in conceptually simple extensions to the Opal kernel and its interface, forms a complete foundation for a distributed operating system for a networked workstation cluster. The network virtual store facilitates cooperation across the cluster and breaks down the dichotomy between local memory and persistent storage. The persistent network capability space extends the global access, caching, and persistence of shared objects to encompass protected objects as well. Persistent and distributed resources use the same format, access methods, and protection as resources in local memory. Users employ standard programming languages to program the information system, augmented with runtime facilities for managing sharing (e.g., transactions) and protection (e.g., protected objects).

Although our current prototype falls well short of our ultimate goals, it provides evidence that such a system can be constructed according to the principles in this chapter, given sufficient address space and integration of supporting work. Basically, we have reduced the problem of a fully functional single address space cluster to two key subproblems: (1) secure network communication, and (2) physical storage management for recoverable and coherent memory. These are core operating system problems; acceptable solutions have been developed and are continually refined by ongoing research.

Chapter 9

Conclusion

The move to 64-bit addressing is a qualitative shift that is far more significant than the move to 32-bit architectures in the 1970s. 64-bit architectures remove basic addressing limitations that have driven operating system design for three decades. On 32-bit architectures, virtual addresses are a scarce resource that must be multiply allocated in order to supply executing programs with sufficient name space. Wide-address architectures can increase the available address space by ten decimal orders of magnitude, making it worthwhile to consider alternative operating system models that exploit large virtual address spaces.

This dissertation has developed and explored a *single virtual address space* operating system model in which the system assigns globally unique virtual addresses to data. We have described the design and implementation of Opal, a single address space operating system for paged RISC architectures. A key principle of Opal is that memory protection is orthogonal to memory addressing -- reflected in the single address space. Virtual addresses in Opal have a globally unique interpretation independent of the context in which those addresses are issued; a given piece of data appears at the same virtual address regardless of where it is stored or which programs access it. Context-independent addressing has three principal advantages:

- It permits sharing of code and data by reference. Stored code and data is meaningful to any execution context in the system. By decoupling protection domains from address spaces, it improves application control over how protection is used. User programs may change protection choices dynamically, or use direct read-only or read-write memory sharing as a lightweight alternative to interaction by messages, streams, or protected procedure calls.

- It extends directly to a uniform single-level virtual store, accessible across a distributed memory machine or a workstation cluster. This allows memory-mapped data structures to be shared across the network or saved on secondary storage in their native virtual memory format. We believe that an extended single address space can meet the storage and sharing needs of complex database applications (e.g., aircraft CAD) in a simple and direct way.

- It eliminates the need to support multiple sets of virtual address mappings in the virtual memory software and memory system architecture. This can simplify use of hashed translation tables that support sparse virtual addressing and enables other architectural enhancements [KCE92, CKD94].

The single address space structure introduces a large number of operating system issues. It affects the way that virtual memory is allocated, reclaimed, made accessible to programs, and bound to physical storage, as well as the organization of code and data within virtual memory. Moreover, the system must provide mechanisms and policies to manage the global virtual memory in order to meet three basic needs.

- **Protection.** Since any program can address any part of the global virtual memory, the system must provide primitives to protect data without negating the flexible sharing made possible by the single address space.

- **Resource management.** Since the global virtual memory is a shared resource, use of virtual memory must be controlled in a way that permits each application or language environment to manage data according to its needs, while isolating environments from misbehavior by other entities in the system.

- **Consistency.** In an extended single address space with distributed or persistent data, the system must provide a consistent view of the global virtual memory across the participating nodes, independent of failures of those nodes. The notion of "consistency" must be flexible enough to meet the needs of diverse applications in the most efficient way.

The principal objectives of our research with Opal were to define and implement (1) system mechanisms that address the issues raised by the single address space approach in a flexible way, (2) a sharing and protection model that can be supported efficiently on standard RISC processors, and (3) higher-level system facilities that allow applications to exploit the simple and efficient sharing made possible by the single address space. For this research I defined the details of the Opal system interface and organization, prototyped the core features of the system on 64-bit hardware, and used the prototype to explore new approaches to organizing applications, using shared memory and memory-mapped long-term storage.

Although the basic Opal mechanisms are general enough to support existing applications with no significant performance costs, our work has been oriented toward improving support for complex application systems composed of interacting components that are independently developed and evolving. The Opal system facilities support a view of integrated systems in which the division of an application system into modular components can be static, but the protection and sharing relationships among those components are fluid. The goal is to allow user code that performs a computation to operate on shared data -- including persistent and distributed data -- efficiently and with minimal changes. Object-based facilities built above Opal show that modular programs may be configured for a range of uses of protection and sharing, with minimal impact on application source code.

Opal is closely related in style and objectives to a number of other systems, both commercial and experimental, dating back over the last 25 years. Dennis and Van Horn's early description of a software system with protection domains, capabilities, and dynamic sharing appeared in the late 1960s [DVH66], and many systems tried to implement that approach. Opal is significant because it exploits modern 64-bit processors to meet the goals of previous systems in a way that is simple, general, and efficient. In particular, it requires no special hardware, supports uniform context-independent naming at the memory address level, includes persistent storage, and can be efficiently implemented in a compatible way alongside current operating systems on current wide-address processors. Opal also embodies a modern division of responsibilities between the hardware, operating system kernel, system services, and language environment. In particular, the elements of Opal's object model are common to a range of systems that support object-based sharing and protection, e.g., capability-based architectures with a single object address space, but in contrast to many of these systems, object support in Opal is implemented entirely in a runtime library on standard hardware, above the system-level abstraction of a flat global virtual memory.

We conclude from the Opal experience that the single address space model is a useful operating system organization that can be implemented efficiently, and will become viable for production use as address space sizes continue to grow. Although it is appealing, the single address space structure is a radical departure from accepted operating system models. The ultimate value of the model will be determined by the extent to which its proven flexibility can be exploited by the designers of integrated application systems. The primary contribution of our work with Opal is to provide the foundation necessary to gain this experience, and an enhanced understanding of the advantages and limitations of both private and single address space structures.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4), 1983.

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 95–109. ACM, October 1991.

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, April 1991.

[AM83] J.E. Allchin and M.S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[APW86] M. Anderson, R. D. Pose, and C. S. Wallace. The password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.

[ARS89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 123–136, December 1989.

[BALL90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[BDA+91] R. Balter, D. Decouchant, A.Duda, A. Freyssinet, S. Krakowiak, and Others. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1), 1991.

[BMvdV93] Alberto Bartoli, Sape Mullender, and Martijn van der Valk. Wide address spaces – exploring the design space. *ACM Operating systems Review*, 27(1), January 1993.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 217–230, December 1993.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[BT88] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[CAC84] W. P. Cockshot, M. P. Atkinson, and K. J. Chisholm. Persistent object management system. *Software – Practice and Experience*, 14(1), January 1984.

[CAL+89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[CFW90] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform object management. In F. Bancilon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology – International Conference on Extending Database Technology 1990 (Lecture Notes in Computer Science 416)*, pages 253–268. Springer-Verlag, 1990.

[Cha90] Jeffrey S. Chase. Persistent object management, July 1990. URL: *http://www.cs.duke.edu/chase/papers/pobject.ps*.

[CKD93] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327, October 1993.

[Cla92] J. D. Clark. *Windows Programmer's guide to OLE/DDE*. Prentice-Hall, 1992.

[CM88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Cus93] Helen Custer. *Inside Windows/NT*. Microsoft Press, 1993.

[Cyp90] Cypress Semiconductor, San Jose, CA. *SPARC RISC User's Guide*, 2nd edition, Feb. 1990.

[DD68] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[Dig81] Digital Equipment Corporation, Maynard, MA. *VAX Architecture Handbook*, 1981.

[Dig92a] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1992.

[Dig92b] Digital Equipment Corporation, Maynard, MA. *DEC OSF/1 Programmer's Guide*, October 1992.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, December 1993.

[DPH92] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Decoupling modularity and protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, May 1992.

[DPSW89] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. The treatment of persistent objects in Arjuna. In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.

[Dra90] Richard Draves. A revised IPC interface. In *USENIX Mach Workshop Proceedings*, October 1990.

[Duc89] Dan Duchamp. Analysis of transaction management performance. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 177–190, December 1989.

[DV66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[DWA94] Michael D. Dahlin, Randolph Y. Wang, and Thomas E. Anderson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 267–280, November 1994.

[Fab74] Robert S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[FBS89] Alessandro Forin, Joseph Barrera, and Richard Sanzi. The shared memory server. In *Proceedings of the Usenix Conference*, pages 229–242, Winter 1989.

[FCL93] Michael J. Feeley, Jeffrey S. Chase, and Edward D. Lazowska. User-level threads and interprocess communication. Technical Report 93-02-03, University of Washington, Department of Computer Science and Engineering, March 1993.

[FCNL94] Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narasayya, and Henry M. Levy. Integrating coherency and recoverability in distributed systems. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 215–227, November 1994.

[FMP+95] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy and Chandu Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Annoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 15–26. ACM, June 1990.

[GMS87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual memory architecture in SunOS. In *Proceedings of the Summer USENIX Conference*, pages 81–95, June 1987.

[GN93] William Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3), July 1993.

[GSB+93] W.E. Garrett, M.L. Scott, R. Bianchini, L.I. Kontothanassis, R.A. McCallum, J.A. Thomas, R. Wisniewski, and S. Luk. Linking shared segments. In *Proceedings of the Winter 1993 Usenix*, January 1993.

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, October 1992.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address space operating system. In *The 17th Annual Computer Science Conference*, January 1994. Australian Computer Science Communications.

[Hew90] Hewlett-Packard, Cupertino, CA. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1st edition, Nov. 1990.

[HH93] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference*, pages 469–480, June 1993.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[HO91] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software – Practice and Experience*, 21(4):375–390, April 1991.

[HSH81] M.E. Houdek, F.G. Soltis, and R. L. Hoffman. IBM System/38 support for capability-based addressing. In *Proc. of the 8th Symposium on Computer Architecture*, May 1981.

[IBM88] IBM. *Application System/400 Technology*. International Business Machines, 1988.

[Int88] Intel Corporation, Santa Clara, CA. *80386 Programmer's Reference Manual*, 1988.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 67–77, October 1986.

[KC94] David Kotz and Preston Crow. The expected lifetime of "single-address-space" operating systems. In *Proceedings of the 1994 SIGMETRICS*, May 1994.

[KCD+81] Kevin C. Kahn, William M. Corwin, T. Don Dennis, Herman D. Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, Fred J. Pollack, and Michael R. Gifkins. iMAX: A multiprocessor operating system for an object-based computer. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 127–136, December 1981.

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, October 1992.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[LCC94] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. Application-controlled physical memory using external page-cache management. In *First Symposium on Operating System Design and Implementation*, pages 153–164, November 1994.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122, November 1987.

[Lee89] Ruby B. Lee. Precision architecture. *IEEE Computer*, pages 78–91, January 1989.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 175–188, December 1993.

[LL82] H.M. Levy and P.H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, pages 35–42, March 1982.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[Mai89] David Maier. Making database systems fast enough for CAD applications. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 573–582. Addison-Wesley, 1989.

[McC91] Thomas McCabe. Programming with mediators: Developing a graphical mesh environment. Master's thesis, Department of Computer Science and Engineering, University of Washington, 1991.

[Mic91] Microsoft Corporation, Redmond, WA. *Microsoft MS-DOS Operating System User's Guide and Reference*, 1991.

[MIP91] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, first edition, 1991.

[Mis90] M. Misra. IBM RISC/6000 Technology, 1990.

[MJLF84] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[Mog94] Jeffrey C. Mogul. Recovery in spritely NFS. *Computing Systems*, 7(2):201–262, Spring 1994.

[MS86] David Maier and Jacob Stein. Development of an object-oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages

472–482, September 1986.

[MS87] Paul R. McJones and Garret F. Swart. Evolving the Unix system interface to support multithreaded programs. Technical Report 21, DEC Systems Research Center, September 1987.

[MT86] S.J. Mullender and A.S. Tanenbaum. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–299, 1986.

[MWO+93] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. Design and implementation of an object-oriented 64-bit single address space microkernel. Technical Report 9, City University, London, 1993.

[Mye93] Andrew C. Myers. Resolving the integrity/performance conflict. In *Proceedings of the Fourth ACM/IEEE Workshop on Workstation Operating Systems*, pages 156–159, 1993.

[Nar95] Vivek Narasayya. An analysis of swizzling costs in an OODBMS, May 1995. URL *http://www.cs.washington.edu/homes/nara/quals/report.ps*.

[Org83] Elliot I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[OSS+92] T. Okamoto, H. Segawa, S.H. Shin, H. Nozue, K. Maeda, and M. Saito. A micro-kernel architecture for next generation processors. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 83–94, April 1992.

[PKW81] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson. The iMAX-432 object filing system. In *Proc. of the 8th ACM Symposium on Operating Systems Principles*, December 1981.

[RA85] J. Rosenberg and D.A. Abramson. MONADS-PC: A capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

[RAA+88] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Fredreic Herrmann, Pierre Leonard, Sylvain Langlois, and Will Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4), 1988.

[RDH+80] David Redell, Yogen Dalal, Thomas Horsley, Hugh Lauer, William Lynch, Paul McJones, Hal Murray, and Stephen Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Ros92] John Rosenberg. Architectural and operating system support for orthogonal persistence. *Computing Systems*, 5(3), July 1992.

[SBG+89] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, and Daniel Yellin. *Hermes Language: Tutorial and Reference Manual*. Springer-Verlag, 1989.

[SBMS93] Margo Seltzer, Keith Bostic, Marshall McKusick, and Carl Staelin. An implementation of a log-structured file system for Unix. In *Proceedings of the 1993 Winter Usenix Conference*, pages 201–220, January 1993.

[SBN82] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Readings and Examples*. McGraw Hill Book Company, 1982.

[Sch94] René W. Schmidt. Exploiting shared memory for protected services. Master's thesis, University of Washington, Department of Computer Science and Engineering, June 1994. CSE 94-06-

03.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[SKN95] Kevin Sullivan, Ira Kalet, and David Notkin. Mediators in a radiation treatment planning environment. Technical Report 95-08-01, University of Virginia, Department of Computer Science, August 1995. Submitted to IEEE Transactions on Software Engineering.

[SLM90] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[SMK+94] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(4):33–57, February 1994.

[SN92] Kevin Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering*, 1(3), July 1992.

[SO92] Avi Silberschatz and Banu Ozden. The shared virtual address space model. Technical Report TR-92-37, University of Texas at Austin, Department of Computer Science, November 1992.

[ST85] Daniel Sleator and Robert Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[SW94] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, June 1994.

[SZ90] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. In *Proceedings of the Fourth International Workshop on Persistent Object Systems: Design, Implementation and Use*, September 1990.

[SZBH86] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 4(8), October 1986.

[Tha86] Satish Thatte. Persistent memory: Merging AI-knowledge and databases. *Texas Instruments Engineering Journal*, pages 151–159, January 1986.

[TK95] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[Tra82] Irving L. Traiger. Virtual memory management for database systems. *Operating System Review*, 16(4):26–48, 1982.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[Wul74] William A. Wulf. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[YBA93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low latency protection in a 64-bit address space. In *Proceedings of the Summer USENIX Conference*, June 1993.

[You89] Michael Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. PhD thesis, Carnegie Mellon University, November 1989. CMU-CS-89-202.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

Vita

Jeffrey Scott Chase was born in Philadelphia, Pennsylvania on December 21, 1962, and grew up in Ithaca, New York. In 1985 he received a B.A. *cum laude* from Dartmouth College in Hanover, New Hampshire, with a double major in Mathematics and Computer Science. He then joined Digital Equipment Corporation's Unix Engineering Group, then in Merrimack, New Hampshire. In 1987 he moved to Seattle and began graduate studies at the University of Washington while still employed by Digital, first as a Digital GEEP fellowship recipient, then later as a Senior Software Engineer at the DECwest engineering group in Bellevue, Washington. He received the M.S. degree in Computer Science from the University of Washington in 1989, and the Ph.D. degree in Computer Science in 1995. He is currently an Assistant Professor of Computer Science at Duke University in Durham, North Carolina.