Mechanisms for Reliable Distributed Real-Time Operating Systems

The Alpha Kernel

J. Duane Northcutt
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ACADEMIC PRESS, INC.
Harcourt Brace Jovanovich, Publishers

Boston Orlando San Diego
New York Austin London Sydney
Tokyo Toronto

ACADEMIC PRESS, INC.
Orlando, Florida 32887

United Kingdom Edition published by
ACADEMIC PRESS INC. (LONDON) LTD.
24–28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data
Northcutt, J. Duane.
Mechanisms for reliable distributed real-time operating systems.

(Perspectives in computing; vol. 16)
Bibliography: p.
1. Operating systems (Computers) 2. Real-time data processing. 3. Electronic data processing—Distributed processing. I. Title. II. Series.
QA76.76.063N67 1987 004'.33 87-1421
ISBN 0-12-521690-4 (alk. paper)

87 88 89 90 9 8 7 6 5 4 3 2 1
Printed in the United States of America

To my brother,
Richard Vernon Northcutt,
1961 – 1979

Preface

This research monograph describes the Alpha kernel — a set of mechanisms that support the construction of reliable, modular, decentralized operating systems for real-time control applications. This work, performed in the course of the author's Ph.D. thesis research, is part of the Archons project at Carnegie-Mellon University, led by Prof. E. Douglas Jensen.

The Alpha effort is aimed at providing fresh experience with operating system kernel design in two senses. First, in the context of operating systems *per se*, many current beliefs and practices have evolved from earlier implementations and have limited validity in contemporary settings. Given that there are few technology-independent lessons to be learned in computer systems research, it is important to frequently re-evaluate some of the basic premises and their system implications. Second, in the context of real-time control systems, operating systems are substantially behind the state of the art, making many deployed systems much less cost-effective than they could be. This creates an attractive opportunity for the infusion of new technology into this area.

The Alpha kernel incorporates a number of carefully integrated, modern techniques, such as: decentralized management of global system (not just local node) resources; kernel level support for atomic transactions and replication; object orientation; judicious exploitation of hardware support; and strict exclusion of policy from the kernel mechanisms. To these, Alpha also adds the use of application-derived time constraints (e.g., deadlines) and relative importance attributes for managing resources to achieve optimal responsiveness and utility for its clients.

The design of Alpha was based on fifteen years of industrial experience with distributed real-time supervisory command and control systems, together with nearly a decade of Archons academic research into the fundamental issues of this class of system. The perspective and scope of the Alpha research is the entire computer *system*, so it is being created entirely from the bare hardware up. Its focus extends from the operating system concepts and interface abstractions, down through all the detailed engineering tradeoffs necessary to achieve cost-effective implementations. The most appropriate forms of synergy between the hardware and kernel designs have been, and continue to be, a driving factor in this project.

The Alpha kernel is not an end in itself; it is a vehicle for exploring new ideas and the foundation for the Alpha decentralized operating system. This monograph provides an initial snapshot of the kernel design and implementation part of this relatively large scale research effort. The Alpha kernel design is complete, and as of this writing, mostly implemented. A major application survey and system evaluation task is under way which will provide detailed feedback to augment the experience already gained. The first version of the Alpha real-time decentralized operating system being constructed on the Alpha kernel is scheduled for completion in 1988. A commercial-quality product version of the Alpha kernel is being constructed by Kendall Square Research Corporation, targeted for their own machine, as well as those of other manufacturers' (beginning with Sun Microsystems workstations).

Acknowledgments

I would like to thank my advisor, Doug Jensen, for allowing me the opportunity to do this work and for providing me with such an fine example of how to do computer systems research. Doug is a good friend and an outstanding researcher, and one could not hope for more from a thesis advisor. Special thanks is due to Martin McKendry, who provided the starting impetus for this effort and whose initial concepts helped shape the work presented here. It was Martin who dragged me, kicking and screaming, into the world of objects and convinced me of their necessity in implementing practical, reliable distributed systems, and for this I am deeply grateful. Martin is missed both technically and personally. Also, Rick Rashid deserves thanks for providing me with much needed guidance throughout the course of this work. Rick's advice has always proved to be sound and I wish that I had taken more of it sooner. It has been my good fortune to work with and learn from these people, who are all members of that small community who put their research concepts into practice by building interesting and significant computer systems.

I would like to thank Ray Clark for the assistance he provided while working with me on the kernel and this document. I doubt it would have been possible for me to complete this work without his help. Sam Shipman also deserves thanks for his contributions to this work. Thanks is also due to Jon Bentley for his guidance over the years and for all the effort he put into reading this and making it a better document.

Thanks is also due to the many people at Sun Microsystems Inc. who helped support this project over the last few years. Their support took the form of equipment donations, agreements to allow us to purchase hardware not in their current product line, access to source code, and access to proprietary technical information about their products. I particularly appreciate the fact that they took the time to help us, despite the fact that they were quite busy creating a successful new company, and our requests frequently fell outside of the mainstream of their market interests.

I would like to thank Dan Siewiorek for providing me with the opportunity to publish my work in this way. Also, I would like to acknowledge and thank Huay-Yong Wang, Chuck Kollar, Bruce

Taylor and Dan Reiner for all of their work in support of Alpha. Larry Slomer also deserves thanks for his help in getting this document printed, as well as for the help he has given to the Archons project in the past. Finally, I would like to thank the Office of Naval Research for taking up our burden as their own and relieving us of the greatest impediment to our research.

This research was sponsored in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274, the US Naval Ocean Systems Center under contract number N66001-83-C-0305, and the IBM Federal Systems Division under university agreement number YA-278067. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of RADC, NOSC, IBM, or the U.S. Government.

List of Figures

List of Tables

1
Introduction

This book documents a set of kernel-level mechanisms that support the construction of modular, reliable, decentralized operating systems for real-time control applications. The perspective and scope of this research is the entire computer *system*, rather than the more narrow focus on a subsystem or algorithm. Consequently, its major contributions extend from programming abstractions and an operating system kernel interface down through the detailed engineering tradeoffs required to create, implement, and cleanly integrate the internal mechanisms. Furthermore, this system also illustrates how the judicious use of hardware support can make possible kernel mechanisms and abstractions that might otherwise be impractical. This is an initial report of on-going research by the Archons project, and much work remains to be done in the use, analysis, optimization, and extension of the kernel. However, many of the initial system objectives have already been validated by the successful design and implementation of the Alpha kernel directly on distributed system hardware.

1.1. Background

The operating system kernel described herein is named Alpha and represents the initial phase of the first implementation effort of the Archons project at Carnegie-Mellon University [Jensen 84]. This project, led by E. Douglas Jensen, is performing research on new concepts and techniques of *decentralized computer* resource management (both in operating systems and architectures) for real-time command and control applications. The Archons project has been active since 1979, sponsored primarily by the USAF Rome Air Development Center, with additional funding from the USN Ocean Systems Center, the IBM Corporation Federal Systems Division, and the Fort Worth Division of General Dynamics Corporation.

The other areas of research in the Archons project that influenced the design of Alpha include: setwise-serializable transactions [Sha 85a], interprocessor communications [Applewhite 81], time-driven scheduling [Locke 86], and the object programming model [McKendry 84a].

1.2. Problem Statement

The kernel described here is intended to support a range of system solutions that effectively meet the requirements of various reliable, distributed, real-time command and control applications. This system environment has a number of characteristics that set it apart from other applications. Furthermore, while other existing systems have been concerned with certain individual aspects, the Alpha kernel currently stands alone in addressing such a comprehensive range of operating system problems posed by this application domain.

The direction taken in this effort was not aimed at providing a facility upon which users may directly create applications, but rather to provide a vehicle for validating the research concepts embodied by the kernel, in addition to a wide range of decentralized operating system and real-time command and control application concepts.

1.2.1. Domain of Interest

This research is focused on the area of distributed systems for real-time command and control applications. The following describes both aspects of this application domain (i.e., distributed systems and real-time control) and the characteristics that set it apart from other problem areas.

1.2.1.1. Distributed Systems

Because of the widely differing definitions of "distributed systems" that abound in the literature, it is important that the definitions used in this document be made explicitly clear. For the purposes of this research, a *decentralized computer* is considered to be a machine that consists of a multiplicity of physically dispersed processing nodes, integrated into a single computer through a native, global *decentralized operating system* [Jensen 78a]. A decentralized operating system manages the system's collective, disjoint physical resources in a unified fashion, for the common good of the whole system. This permits the system's clients to be presented with the view of a single machine, capable of being entirely applied to a single application.

A logically singular, yet physically dispersed computer is very important in many contexts, particularly real-time command and control, where the concept of independent users does not exist as it does in typical timesharing and computer network systems [Thomas 78]. The whole system is dedicated to performing a particular mission, and can be thought of as having a single user comprised of the physical processes being controlled.

A decentralized operating system consists of replicated copies that constitute the native operating system on all the system's processing nodes. However, the system's resources are managed in a global fashion. This is done either by direct coordination among the various replicas of the kernel, or locally by each kernel replica as consequences of the higher-level resource management decisions.

These system characteristics are quite different from those of more traditional multiprocessors, computer networks, and other systems with similar hardware structures. For example, in computer networks the resources at a particular processing node are managed by the operating system local to that node, and the collection of autonomous, local operating systems interact in limited ways (e.g., to support such application purposes as file sharing, mail, remote login, etc.) [Lampson 81]. Furthermore, the emphasis in the work described here is on multicomputer systems which do not have shared primary memory, separating it from operating system efforts performed on multiprocessor hardware [Jones 79, Ousterhout 80, Wulf 81].

While there are not yet any instances of a full-functionality decentralized operating system, the work described here is an appropriate kernel subset of one which is expected to greatly facilitate the creation of the remainder of such a system. The Alpha kernel is referred to as a *decentralized operating system kernel*, to differentiate it from both a full-functionality decentralized operating system and conventional network, or shared memory multiprocessor, operating systems.

1.2.1.2. Real-Time Command and Control

The computer systems of interest in this research are used to control a collection of physical processes whose states are sensed and altered by the computer system. The state of these processes changes independently as a result of the external environment, which is not completely under the control of the computer system.

The type of control involved here is *supervisory control*, as opposed to low-level, synchronous sampled data loop functions like sensor/actuator feedback control, signal processing, etc. Supervisory control is a middle-level function, above the sampled data loop functions and below the human interface/management functions. This type of system does not do much direct polling of sensors and manipulation of actuators, nor does it provide extensive man/machine interfaces; rather, it deals with subsystems which provide those functions. Some tasks in a real-time command and control system are periodic and are bound to process activity rates; but most are aperiodic with stochastic parameters, and associated with external stimuli and the interactions of the system with low-level control subsystems [Boebert 78]. The real-time response requirements of a supervisory control system are closer to the millisecond than either the microsecond or second ranges.

Supervisory real-time command and control systems are found in plant (e.g., factory or refinery) automation, vehicle (e.g., airborne, aerospace, or shipboard) control, and surveillance (e.g., air-traffic control) systems.

1.2.2. Special Requirements

A number of important requirements are implied by the real-time command and control application domain. Some of these requirements are unique to this context, and others, while generally applicable to a wide range of systems, are especially important here. It is these application-derived requirements that form the basis from which the programming abstractions supported by the kernel, and the design and implementation of the kernel mechanisms, are derived.

1.2.2.1. Distribution

Integrating physically dispersed hardware into a logically unified system at the operating system level means that the clients of a decentralized operating system should not be required to be aware of, much less manage, the physical dispersion of hardware and software, nor the many complex consequences thereof. These issues should not be allowed to distract the clients from performing their application tasks. In addition, problems of distribution solved at the operating system level represent a non-recurring cost that is not passed on to each application builder.

One of the major functions of decentralized operating systems and network operating systems is to manage the inter-node communication resources for the clients. But in a distributed system, it is highly desirable that physical dispersal of the underlying hardware be made transparent at a low level in the system (the kernel). In this manner, both the system and the application programmers benefit from features such as physical-location-transparency, in a way similar to how programmers benefit from a system-provided process abstraction. However, there are a number of cases (e.g., work assignment, redundancy management, specialized function location, or diagnostics) in which it is appropriate to provide clients of a kernel with information concerning physical location.

In a distributed system, system and application software must be based on *thin-wire* [Metcalf 72], as opposed to shared memory, interconnection techniques. This suggests that performance, availability, and reliability would suffer if system mechanisms make use of centralized structures. Furthermore, the physical dispersal of the system hardware introduces variable and unknown communication delays that exacerbate the already difficult task of attempting to ensure deterministic behavior of the system. The kernel mechanisms for decentralized systems must therefore be adaptive and deal

explicitly with the effects of physical dispersal — i.e., inaccurate and incomplete information. Although it is below the kernel level and deals with a small number of simple, static resources, the Arpanet routing algorithm [McQuillan 80] provides an example of this type of behavior.

It should be noted that the hardware of a decentralized computer system, as defined here, need not be different from that of typical local area networks (i.e., a collection of processing elements with private memory and local peripheral devices, interconnected by an interconnection subnetwork). It is usually the operating system software alone that separates decentralized computer systems from local area networks.

The class of distributed real-time command and control systems of interest to this research involves on the order of 10 to 100 processing nodes, physically dispersed across a distance on the order of 100 to 1000 meters.

1.2.2.2. Reliability

In the context of this effort, reliability means as the degree to which the application goals continue to be met in the face of failures (which is meant to include *faults*, *errors*, and *failures* [Avizienis 78]).

The nature of the physical processes being controlled in real-time command and control systems is usually such that the safety consequences (in terms of personal or property damage) of not managing them properly can be quite severe. Furthermore, it is often the case that the controlling system cannot be maintained easily. Should a component fail, it can be quite difficult (if not impossible) to gain access to the system to perform the necessary repairs, and frequently repairs must be made without interrupting the normal functioning of the system. The *mission* times — i.e., the period of time when the system must continue to provide correct service, uninterrupted by either maintenance or repair activities — of real-time systems can range from hours to years. Thus, the reliability of the system is of utmost importance, even to the point of being more significant than the cost or performance of the system.

The correct and timely execution of a real-time command and control application is typically more important under exception conditions than in normal cases. Also, exceptions (such as hardware failures due to physical damage) in such systems tend to be clustered in both time and space. These characteristics have a significant impact on the nature of the fault tolerance and recovery techniques used in such systems, and are contrary to the premises underlying almost all non-real-time computing system approaches (e.g., RISC-style operating system philosophies and the "end-to-end argument") [DRC 86].

To support the overall reliability goals of a distributed real-time command and control application, the system software must itself meet a certain level of reliability. In addition to this, the system must provide mechanisms that allow suitably reliable applications to be constructed. The kernel should not dictate a specific kind and degree of reliability, but rather it should allow its clients to choose what is desired for each individual set of circumstances, at a cost that is appropriate. Such a flexible, mechanism-based approach is conceptually more difficult to design, but supports increased system efficiency, in that the kernel's clients are free to make the appropriate cost/functionality tradeoffs on a case-by-case basis within their application programs.

The distributed real-time command and control application domain calls for a set of kernel mechanisms that support the following reliability concepts: correctness of actions, high availability of services, graceful degradation, and fault containment. While these concepts are useful in almost any system, they are critical in this application domain.

The correctness of the actions performed by an application is a function of time, sequencing, and completeness — i.e., the correctness of a set of actions is defined by the amount of time it takes each action to execute, the order in which actions execute, and whether, at the end of the set of actions, all of them were successfully executed.

The availability of services is defined to be the extent to which each service remains available to clients across system failures. Typically, a service that is statically and uniquely bound to a particular hardware functional unit becomes unavailable should that hardware unit fail. To increase the availability of services, failures that may result in service disruption must be eliminated, or the means must be provided for resuming the service elsewhere when a failure occurs. The dynamic nature of the physical processes being controlled implies that a great deal of information in a distributed real-time control system degrades with time. This dictates that the availability of information is at least as important as, if not more important than, its consistency. Under some circumstances in real-time applications there is little value in maintaining the consistency of a database if the information is unavailable for use when needed. Similarly, when the correctness of the information in a database degrades over time, there may be little point in attempting to restore it to a meaningful consistent state following a period of unavailability.

Graceful degradation is defined to be the property of a system that permits it to continue providing the highest level of functionality possible as the demand for resources exceeds its currently available capacity. In the context of this work, whenever contending requests for resources cannot all be met

in an acceptable time, the contention should be resolved in favor of the functions that are most critical to the objectives of managing the physical processes being controlled. Information concerning the relative importances of individual application tasks can be provided only by the application programmer. To provide graceful degradation, the system uses this importance information to determine which tasks' needs will not be met, to sustain the highest and most useful level of functionality under overload conditions.

Fault containment is defined to be a property that inhibits the propagation of errors among system components. If a failure occurs (or is induced) in a system or application component, the kernel mechanisms should limit (or assist in limiting) the extent to which the failed component can adversely affect the behavior of others. For example, the kernel should not permit a failed software component to arbitrarily modify the state of other components, either intentionally or by accident. Furthermore, for this property to hold, mechanisms must ensure that a failed software component cannot consume resources in an unconstrained manner and thereby interfere with other, presumably good, software components.

1.2.2.3. Timeliness

The timeliness of the activities performed by a real-time system is considered part of the definition of correct system behavior. It is not sufficient to ensure only that data is correct and consistent, if it is not also presented in a timely manner. In a real-time system, computations must be performed in accordance with the time constraints imposed by the physical processes being controlled, and because of the dynamic nature of the external processes, the value of information degrades with time. A kernel that is to support real-time applications must provide mechanisms which take these time-related issues into account and help application programs in meeting their time constraints [Jensen 76, Wirth 77].

To deal with timeliness concerns, real-time command and control systems frequently attempt to constrain their applications to behave in a highly deterministic fashion — e.g., ensuring that there are always sufficient resources to satisfy all requests, and that all functions always take the same amount of time to complete. This generally is feasible only for certain, very stylized, computations (such as signal processing), and typically at the cost of committing excess resources (which are underutilized most of the time). But for larger and more demanding distributed real-time command and control systems, it is less practical to attempt to pose such constraints because their behavior is necessarily much more dynamic and stochastic. It is, however, quite difficult to construct a system that is adaptive and yet able to meet demanding timeliness constraints because adaptive distributed algorithms are difficult to create and tend to be costly in terms of performance.

An alternative approach (used in Alpha) is to have clients provide both run-time and compile-time inputs to the kernel that are used in resolving contention for resources. Examples of the type of information that may be supplied to the system include: an indication of the relative importance of each computation, the expected completion times for various services, the deadline for completing an activity, and whether there is any value in performing the computation once the deadline has passed. Using this approach, the system attempts to meet all the time constraints, adapts to unexpected events, and, when the demands exceed the supply of system resources so that not all time constraints can be met, discards requests according to some client-specified policy (e.g., discard the least important ones first, or discard the ones that maximize the number of requests whose time constraints are met).

1.2.2.4. Modularity

In real-time command and control systems, modifications, technology upgrades, testing, maintenance, and other life-cycle items invariably make up the most significant portions of a system's cost [Savitzky 85]. Typically, the software-related costs predominate because the requirements are poorly understood at system design time and continue to evolve throughout, not just the design and implementation phase of the system, but even during the system's lifetime (which may be a decade or more) [Boehm 81, Lehman 85].

This implies the need for system software to provide a programming model that supports such desirable software engineering attributes as modularity and maintainability. Because these attributes are more frequently associated with languages than operating systems, a kernel for real-time command and control might facilitate modularity by taking into account the run-time packages of languages that address these attributes of modularity.

In the realm of real-time command and control control, system sizes range from quite small (e.g., an individual vehicle), to very large (e.g, an entire plant). Furthermore, in a decentralized system, processing nodes may be added and removed (either statically off-line, or dynamically due to the run-time failure and recovery of hardware). The operating system must itself be able to function effectively across a wide range of application and system sizes to take advantage of the opportunities for extensibility offered by the inherently modular hardware architecture of a distributed system, and to provide the reliability available from reconfiguration.

The properties of modularity and extensibility are not unique to decentralized computer systems. Centralized systems may provide the necessary level of performance for a given application and may

incorporate system software that is quite modular and extensible; however, centralized uniprocessor and multiprocessor hardware pose quite severe limitations in these respects. Decentralized systems offer a wider range of cost/performance choices than is usually available from a family of centralized processors or shared-memory multiprocessors. Overall, the physical properties of decentralized systems offer the potential for greater reliability, extensibility and modularity, both within the system as well as in the applications [Franta 81].

Modularity and extensibility (even more so than reliability) are considerably more difficult to quantify and measure than other system attributes, such as performance. This may account for the fact that concern for performance (usually in the form of throughput) often dominates other considerations, fostering the misperception that these other system attributes are of lesser importance. This is unfortunate since system performance increases are derived automatically from advances in semiconductor technology, while increases in system reliability, modularity, and extensibility result almost exclusively from thoughtful effort by system designers.

1.2.3. Current Practice

Operating systems in existence today do not adequately address the requirements posed by the type of distributed real-time command and control applications described here. While some systems deal with certain aspects of the overall problem, it is safe to say that none meets even a significant number (much less all) of them.

1.2.3.1. Distribution

Despite the fact that physically dispersed systems are becoming more wide-spread, virtually none of them attempts to be globally unified in the manner described in this work (see Subsection 1.2.1.1). Most distributed systems are simply networks, or at most federated (dedicated mission) systems, functioning as loosely associated collections of autonomous computing nodes with the ability to interact with each other [Thompson 80, Lampson 81]. In typical distributed systems, each node manages its local resources independently, and the client's interface to the system is usually non-location-transparent and non-uniform.

1.2.3.2. Reliability

The most common means of attempting to provide reliability in real-time control systems is through the extensive use of excess assets — i.e., the system is provided with far more resources than necessary to meet the application's (steady-state) computational demands in order to ensure that the system's objectives can continue to be met in the face of failures. These systems rely on a very low level of average system resource utilization in order to achieve their reliability goals (e.g., in some cases resource utilization can be no more than about 70% in order for the system to meet its specified guarantees [Lechoczky 86]).

Most extant systems that address the reliability issues described in Subsection 1.2.2.2 tend to use low-concurrency, high-cost consistency maintenance schemes (e.g., atomic transactions), or equally costly redundancy management techniques (e.g., replication). The use of these techniques has not been directed towards the area of distributed real-time command and control. Frequently (regardless of the techniques used) a system's reliability techniques impose substantial constraints on the system and application programmers. Many systems that have been constructed to explore reliability techniques emphasize only one aspect of system reliability. They also tend to impose fixed costs, irrespective of whether the client currently needs the system's reliability services.

1.2.3.3. Timeliness

There is a range of degrees to which a system can support the needs of real-time applications. A system could make it difficult to meet real-time demands (e.g., by trying to enforce a "fairness" policy in resolving contention for resources), or it could provide mechanisms that help in the construction of real-time applications, or it could be structured so as to make real-time guarantees for its applications.

Most so-called real-time operating systems today (e.g., [Ready 86]) have relatively little that separates them from other types of operating systems. For the most part, they provide minimal functionality, preferring instead to pass the time, space, and intellectual complexity burdens of system resource management on to the application programmer. These little executives strive to avoid doing anything that would make it difficult for applications to meet their time constraints, and try to provide service to the clients in a predictable fashion (primarily with respect to time); they incorporate little more than priority interrupt handling and context swapping facilities in an attempt to facilitate real-time responsiveness.

Practical real-time applications call for more than just raw performance. This is despite the fact that a sufficiently high degree of brute force hardware performance alone would usually be sufficient (in

principle) to meet the computational (if not cost, size, weight, and power) needs of virtually any real-time applications. Current real-time operating systems tend to abdicate their system resource management responsibilities at the expense of the application programmers, thereby increasing a system's implementation costs and degrading its performance.

Conventional practice in real-time computing systems today is to attempt to resolve instances of contention for system resources through the use of simple, static priority schemes. These systems provide only the priority mechanism; the difficult task of making the priority assignments is left to the client, and typically requires a great deal of *tuning*. It can easily be shown that, in general, fixed priority assignments are incapable of meeting activity deadlines, even when the computing capacity is far greater than the needs of the activities. Experience is consistent with this principle: given certain (often rather unrealistic) assumptions, the frequently used rate-monotonic scheduling technique is known to be an optimal static algorithm with respect to meeting periodic deadlines; otherwise, only trivial systems manage a successful balance between priority responsiveness and resource utilization. Despite the fact that dynamic priority assignments can significantly out-perform static ones, they are rarely used in actual real-time systems.

Some real-time systems attempt to guarantee that all of an application's timeliness requirements are always met. While it would be ideal for a real-time operating system to provide such timeliness guarantees, current approaches to making them introduce *significant* programming constraints and usually require that the system conform to certain unrealistically over-simplified assumptions concerning its behavior (e.g., as in [Leinbaugh 80]). Most efforts in this area focus on providing high resource utilization for low-level (i.e., closed-loop, sampled data) control applications where tasks are deterministically periodic and have no value to the system if their deadlines cannot be met (i.e., tasks have *hard deadlines*). The appeal of such approaches is their analytical tractability, but they are not suitable for more general real-time command and control contexts, which are characterized by predominately aperiodic tasks and are less amenable to such rigid and stylized treatment. In general, it is currently no more practical to make absolute timeliness guarantees for general, unconstrained real-time systems than it is practical to prove the correctness of large, complicated programming systems.

1.2.3.4. Modularity

While certain existing systems have emphasized modularity and extensibility in their designs, they have not been systems of the type that is of interest in this research (see Chapter 7). Although some real-time control systems exhibit a significant degree of architectural modularity (via the use of processing nodes interconnected by high-performance communication subnetworks [Jensen 78b]), and others provide modularity at the programming language level (via a high-level language such as Ada), there are few examples that have explicitly attempted to be usefully modular in their system-level design.

There are operating systems that provide a high degree of modularity and extensibility at their interfaces and within their structure, however few of these have been distributed systems and fewer still have been for real-time command and control. Currently, most real-time systems attempt to make the behavior of the system and applications as static and deterministic as possible, and attempt to validate the correctness of the system through exhaustive testing [Quirk 85]. A common characteristic of this type of system is the high cost associated with the addition or modification of system functionality, and the inability to cope with unanticipated behavior [Parnas 77, Glass 80].

1.3. Technical Approach

This research was directed toward the synthesis of new concepts to meet the particular needs of the previously defined application domain. This required a carefully integrated software/hardware approach, along with the exploration of the tradeoffs required to implement those concepts effectively. One of the results of this research was the creation of a set of programming abstractions that are intrinsically well suited to modular, reliable, decentralized operating systems, and the design and implementation of a set of kernel-level mechanisms in support of them.

1.3.1. Systems Research

This effort is aimed at deriving fresh experience with decentralized operating system kernel design in a modern context. This is important because a great many current operating system beliefs and practices are derived from earlier implementation efforts and have limited validity in contemporary settings. Given that there are few technology-independent lessons to be learned in systems research, it is important to reevaluate some of the basic premises of systems design through actual design and implementation efforts.

It seems clear that computer systems research must be validated by empirical studies — it is impossible to do credible systems research without actually building and using systems [Eastport 85]. While simulation may be adequate for some types of algorithm studies, it is not an effective means of validating nontrivial systems research. Real computer systems (especially decentralized ones) tend to be too complex to be modeled accurately, and there are far too many technology-driven aspects of system construction that must be abstracted out in even the most detailed simulations. Emulation of systems, by building application programs on top of other systems, is equally insufficient. There is no substitute for actual experience in the systems world, and a true engineering effort is required in order to make reasonable tradeoffs and to introduce reality into a systems research project.

The systems research concepts described here are validated through the construction of an operating system kernel on bare hardware — not by the commonplace compromise of modifying or building on top of an existing operating system such as UNIX. This approach required that a large number of details (some of which might be considered peripheral issues) be addressed, and a number of diverse concepts carefully merged in order to create a functioning system. In paper studies and application-level emulations, most of these interactions are not considered and many low-level issues are dismissed as supposed "engineering details." This unfortunate attitude limits and distorts the results, in that some such details have a substantial effect on the conceptual nature of the operating system (such as its overall structure and programming abstractions). Furthermore, some low-level details may conflict with the underlying assumptions of the high-level work, thereby invalidating the results. It is because the systems area has few concepts that hold true regardless of low-level, technology dependent details, that the empirical validation of systems research concepts is so critical.

Building an operating system kernel directly on bare hardware provides the opportunity to explore systems issues without having the results perturbed by the policies and implementation artifacts of an underlying host operating system. It also assures a sufficiently high degree of performance to allow realistic applications to be constructed that help illustrate the system's behavior under representative loading conditions. The efforts of others have made abundantly clear the impracticality of implementing such facilities on top of existing, conventional operating systems [Spector 84, Almes 85]. The experience derived from implementing the Alpha kernel mechanisms thus far indicates that the effort required to construct a kernel on bare hardware is significant, but not unreasonable in comparison to the amount of effort that must otherwise be expended in attempting to overcome the impediments to performing effective systems research on top of an inappropriate, preexisting operating system base.

1.3.2. Experimentation Environment

As with all empirical work, this research was performed within a framework of constraints. Some restrictions were applied in order to focus this effort on aspects of the problem that were considered most interesting or important, and to narrow the effort down to a level at which initial results could be obtained in a reasonable amount of time.

Because this research is in the area of operating systems and not programming languages, the constraint of using an existing language was adopted. The C programming language was chosen for system implementation and application coding because it is sufficiently expressive for low-level system-programming, and it is well-supported by the project's development hardware (i.e., Sun Microsystems workstations). Despite the lack of a language effort, the Sun Microsystems/UNIX development environment made it straightforward to create simple pre-processing tools that provide an enhanced programming interface to the primitives of the Alpha kernel.

To focus the emphasis of this research more tightly, a decision was made to implement the kernel mechanisms (to the greatest extent possible) on conventional off-the-shelf hardware, in a configuration that could be considered representative of current local area network systems.

The Alpha kernel mechanisms execute on a loosely-coupled collection of processing nodes constructed from largely off-the-shelf system components (i.e., the Archons testbed [Clark 83]). The testbed facilitates the development of system software by a collection of system programmers, working from individual (remote) workstations. Furthermore, the nodes of the testbed were designed to allow the exploration of various operating system concepts that may benefit from hardware support.

To effectively emulate special-purpose hardware in support of operating system functions, the testbed nodes are implemented as simple shared-memory multiprocessors. Each node consists of a collection of processing elements (i.e., processors with local memory and I/O devices) in a common backplane. Additionally, a global interprocessor interrupt generation mechanism and a globally shared memory form the basis for the interprocessor communications service within a node (see Figure 6-2). This hardware configuration does not imply that the kernel mechanisms developed here are intended for general-purpose multiprocessor nodes — that extension is one of our future plans. Rather, we currently use general-purpose multiprocessor nodes to emulate processing nodes in which special-purpose processors are dedicated to specific operating system functions.

Each processing node in the testbed consists of a number of logical hardware units, including: an

Application Processing Element, on which the application and much of the kernel executes; a *Scheduling Processing Element*, where the application processor's scheduling facility executes; and a *Bus Interface Unit*, consisting of a *Communication Processing Element* that carries out much of the inter-node communication functionality of the kernel, and a *Network Interface Controller* that provides the low-level interface to the communication subnetwork. Also, some nodes have one or more 84MB disk drives attached to them. Such nodes have two additional hardware units, namely the *Secondary Storage Processing Element*, which carries out the kernel's secondary storage functions, and the *Disk Controller*, to which the node's disk drives are attached. Testbed nodes may also have various special-purpose I/O devices (e.g., displays, and sensor/actuator units) attached to them via interface boards.

The processing elements are custom-modified Sun Microsystems single board computers [Bechtolsheim 82] — consisting of a 10Mhz MC68010 processor, 1MB of local read/write memory (expandable to 8MB), a memory management unit, and a complement of on-board devices (i.e., dual UART, programmable timers, etc.). The nodes are interconnected by a private 10Mb Ethernet, and the testbed itself is connected to a department-wide 10Mb Ethernet via a Sun Microsystems workstation used as a console interface processor and a network gateway. Figure 6-1 is a representation of the Archons Testbed on which the Alpha kernel is being developed.

1.3.3. Kernel Abstractions

The most visible manifestation of this research effort is the collection of *mechanisms* with which the programming abstractions of the Alpha kernel are implemented. The mechanisms described here do not constitute a full operating system, but rather an operating system *kernel*. The purpose of a kernel is to provide fundamental abstractions and mechanisms that support a range of different system interfaces (i.e., operating systems and languages) similar to [Habermann 76]; which is not the same as a trivial operating system or an executive. The Alpha mechanisms are carefully and deliberately devoid of policy decisions, and are meant to support the exploration of a wide range of decentralized operating system policies. The interface provided by the kernel is not (necessarily) intended to be the same interface presented by the operating system to an application programmer.

The abstractions provided by the kernel are based on a combination of the principles of object-orientation [Bayer 79], atomic transactions [Bayer 79], replication [Randell 78], and decentralized real-time control [Lampson 81]. The Alpha kernel interface presents its clients with a set of simple and uniform programming abstractions from which reliable real-time control applications

may be constructed. The kernel mechanisms support an *object-oriented* programming paradigm, where the primary abstractions are *objects*, *operation invocation*, and *threads*.

In addition to the attributes of modularity, information-hiding, maintainability, etc. normally associated with an object-oriented programming paradigm [Cox 86], the programming model described here is especially well suited for the support of decentralized, high-concurrency implementations of the major reliability techniques supported by Alpha (i.e., atomic transactions and replication).

At the highest level of abstraction, objects in the Alpha kernel are equivalent to abstract data types. Objects are written by the application programmer and are similar to Ada packages [Ada 83]. Objects are written as individual modules composed of the specific operations that define their interface. While there currently does not exist an object-oriented programming language for Alpha, a preprocessor provides the programmer with a set of simple language constructs for the composition of objects. This object model used in Alpha emphasizes a simple and uniform interface, with as few specialized artifacts as possible introduced into the programming model. The object abstraction in this kernel extends to all system services, and encapsulates all of the system's physical resources, providing clients with object interfaces to all system-managed resources (i.e., memory, devices, etc.).

All interactions with both user and system objects is via the invocation of operations on objects. The operation invocation mechanism is the fundamental facility on which the remainder of the Alpha kernel is based (this is analogous to the role that the interprocess communication facility plays in Accent [Rashid 81]). The invocation of operations on objects is controlled by the kernel through the use of a capability mechanism. In this way, the ability of objects to invoke operations on other objects can be restricted to only that set of destination objects explicitly permitted. Capabilities can be given to objects when they are created, or they can be passed as parameters of operation invocations. In the kernel, the capability mechanism provides basic, defensive protection at a low cost in performance.

Threads are the run-time manifestations of concurrent computations — they are the unit of activity, concurrency, and schedulability in the kernel. The thread abstraction is similar to the commonly used notion of *process*, except that threads may move among objects via operation invocations, regardless of the physical nodes on which the individual objects may reside. Threads execute asynchronously with respect to each other, allowing a high degree of concurrency but necessitating that the kernel provide a set of concurrency control mechanisms. These mechanisms allow the necessary degree of concurrency control to be applied at a reasonable cost in terms of overall system

performance (i.e., the mechanisms perform their functions quickly and their use does not seriously restrict concurrency in the applications).

With threads it is possible to implement a wide range of system-level control policies, ranging from low-concurrency structures (such as *monitors*) to medium and high concurrency ones. Furthermore, the thread abstraction simplifies the task of time management in the kernel by being a run-time manifestation of client-defined computations. Threads are also more efficient than most process- and message-based client/server model implementations, because each step in the computation does not necessarily involve an interaction with the system scheduler. The thread abstraction maintains a correspondence between the programmer's view of a logical computation, and the system's manifestation of these computations. This feature makes it possible for the client programmer to associate attributes (such as importance, urgency, etc.) with computations. This information can then be used by the system to resolve contention for resources on a global basis and according to a chosen policy.

1.3.4. Kernel Subsystems

Beneath its interface, the kernel uses a collection of relatively conventional facilities to support the Alpha client-level abstractions. Objects are implemented as contiguous extents of virtual memory, mappable within an address space. Object operations are execution entry points within these address spaces. At any node in which a thread is active, it is manifest there in a fashion similar to that of conventional processes. The manipulation of virtual memory hardware and the exploitation of the functionality provided by a programmable network interface unit (i.e., the communication subsystem) permit a thread to move efficiently among objects.

The techniques used to support the kernel mechanisms include such common kernel facilities as inter-node communications, application processor scheduling, memory management and secondary storage management, in addition to the effective (and somewhat non-standard) use of system hardware.

The functionality of the Alpha kernel is provided through an implementation that employs largely "standard" (i.e., process-oriented) techniques of system software design, despite the fact that the kernel provides a "non-standard" (i.e., object-oriented) programming model to the clients. In large part this is because most currently available hardware is designed to support a process-oriented programming model (e.g., the Sun Microsystems processor boards are well suited to support the UNIX operating system).

This research does not so much represent an attempt to take advantage of some particular hardware (e.g., hardware that supports capability-based addressing [Wilkes 79] or some object model [Cox 83]), as it does an attempt to explore a set of system software concepts. Therefore, it is considered best to remain as independent of the specifics of the underlying hardware as possible until the conceptual issues being explored are better understood. When the costs and benefits of the abstractions themselves are known it may be appropriate to suggest hardware structures to support them. A rational viewpoint on some of the issues involved in hardware support for a particular object model appears in [Colwell 85].

While the kernel is designed to execute on standard, off-the-shelf processing elements, the judicious use of hardware makes possible functions that ordinarily would be considered impractical. The general shared-memory multiprocessor structure of the testbed nodes allows the emulation in software of specialized hardware support for various kernel functions. The major functions within the Alpha kernel supported by adjunct processing elements are: inter-node communication, secondary storage management, application processor scheduling, atomic transactions, and object replication. This research is intended to demonstrate that significant functionality and performance benefits can be achieved through the use of additional points of hardware control in support of actual concurrency within the kernel.

1.3.4.1. Inter-node Communications

The need for applying some form of hardware support for inter-node communications has been apparent for some time. Previous system efforts, of our own [Jensen 78a] and others [Farber 72, Wittie 79], along with early experience with the requirements of systems similar to the Alpha kernel [McKendry 84a], have illustrated the potentially high cost of communication to provide coordination among nodes in a distributed system. Thus, the Alpha kernel contains a programmable, concurrently executing network interface unit dedicated to the management of the communication resources at a node. The communications processing element in each of the Alpha testbed nodes off-loads the communications overhead from the application processor and makes it practical to support highly functional forms of object operation invocation semantics (e.g., low-level support for the system's atomic transaction and replication services).

The use of hardware communication support makes it possible to provide functions beyond the simple ability to be responsive under high network loading. Included in these functions are logical destination addressing and specialized low-level protocols. Logical destination addressing implies that the destination of each instance of inter-node information interchange is a logical (i.e., software-

defined) entity, as opposed to a physical entity (i.e., a specific processing node) [Mockapetris 77]. Some form of logical addressing is common at the higher levels in system and application software, but is usually not available to the operating system itself, where it may be just as useful.

Logical addressing supports location independence, since the system is not required to maintain name-to-address mapping tables, nor must it keep them consistent in the face of the dynamic relocation of addressable entities. Furthermore, a comprehensive logical addressing scheme allows system to carry out direct communications with logical entities (e.g., transactions, objects, or threads). This feature simplifies many of the communication-related tasks within the kernel, and also lends support to the kernel's object replication mechanisms in that multiple objects (at potentially different locations) may be addressed with the same logical name. Because logical addressing is very valuable to decentralized operating system kernels, it should be performed by either the communication subnetwork interface hardware (as in DCS [Farber 72] and HXDP [Jensen 78a]), or within the kernel. In Alpha the latter is done, due to the constraint of using off-the-shelf hardware, which typically supports only physical destination addressing.

The use of programmable network communications controllers in the Alpha processing nodes also allows sophisticated protocols to be supported efficiently at low levels within the system. Such functions as logical clock synchronization and node keep-alive coordination can be performed by the communications unit, without consuming application processing cycles. On the other hand, it is facilities such as atomic transactions and object replication that benefit the most from this low-level support. In the case of atomic transactions, two-phase commit protocols are provided by the communications controller in addition to protocols that provide orphan detection and elimination. For object replication, the communications controller can execute *bidding protocols* [Smith 79], as well as various *consensus schemes* [Gifford 79, Herlihy 86]. These functions are fairly involved and would incur a major performance cost if the application processor were solely responsible for them.

Because of the large amount of functionality that the Alpha kernel embeds within the operation invocation facility, it is important that the communications subsystem exhibit a high degree of performance and flexibility. To this end, the communication subsystem in Alpha has been designed in a rather unusual manner. The communication subsystem software provides a virtual machine which executes communication protocols implemented in terms of state machine descriptions. This approach allows complicated protocols to be developed and modified at low cost, while still maintaining a high level of performance.

1.3.4.2. Virtual Memory Management

The specific aspects of the hardware underlying the kernel that provide support for virtual memory management are an application processor that permits instruction restart and a Memory Management Unit (MMU) that provides for address validation and translation.

While virtual memory is not required for the Alpha kernel, it does provide a number of benefits. In Alpha, virtual memory hardware is used to provide separate address spaces (or contexts) that enforce protection and separation among objects (in support of fault containment). Virtual memory facilities also serve to increase the utilization of the physical memory, as well as supporting the invocation of operations on objects, atomic transactions, and various performance optimizations (e.g., copy-on-write, delayed-allocation, or on-demand-zeroing). In Alpha, virtual memory is primarily considered to be a programming convenience, and not a mechanism for extending physical memory.

Each thread in Alpha is associated with an individual virtual address space (or context). As a thread moves among objects by making a succession of invocations, the objects upon which operations are invoked are mapped into and out of the thread's context. The memory management hardware makes it possible for threads to move among objects efficiently on invocation. When a thread invokes an operation on an object, the currently executing object is mapped out of the thread's context and the new object is mapped in place of it — in effect, performing a partial context swap. The MMU also allows large amounts of information to be passed efficiently among objects in separate (protected) address spaces, as well as among the application processor and the devices within a node (i.e., network interface unit, disk, display devices, etc.).

The *page fault* exception mechanism allows kernel routines to be invoked when certain pages of an object are accessed. This exception mechanism is useful not only in its traditional role of extending physical memory, but can also be used to signal information concerning the access patterns of the application code to the system. For example, the page fault exception allows a range of lazy-evaluation-style optimizations, such as copy-on-write [Fitzgerald 85], to be used to increase system performance.

1.3.4.3. Application Processor Scheduling

The Alpha kernel supports multiprogramming — i.e., the multiplexing of computations onto a single application processor, giving them the appearance that they are executing in asynchronous concurrency with respect to each other. The kernel provides a facility for the binding of threads to application processors in an order specified by the system's scheduling policy.

The kernel's application processor scheduling facility is important to the ability of the system to meet timeliness constraints. To this end, the kernel provides mechanisms that allow the client to specify the timeliness constraints (in the form of deadlines), processing requirements (in the form of expected computation time), and the relative importance of each application activity. This information is used by the kernel for deriving thread schedules, and for detecting and dealing with processing *overloads*. Overload conditions occur when there is contention for application processing cycles (i.e., when there are not sufficient processing cycles to meet all of the application's needs). The kernel is designed to permit the easy substitution of differing contention resolution policies for managing application processing cycles.

A major effort in the area of time-driven scheduling has been underway as part of the Archons Project [Locke 86], and results from this effort have been reflected in the design of the Alpha kernel's scheduling facility. Also, the kernel has been designed to use a separate processing element to allow various high-performance, but potentially complex and cycle-intensive, time-driven scheduling algorithms to be executed concurrently with the application code. The use of hardware support for the scheduling function provides concurrency within the kernel, and reduces the amount of system functionality that burdens the application processing elements.

1.3.4.4. Secondary Storage

In the Alpha kernel, the secondary storage subsystem serves three major functions: to provide a load source for initial program loading and restart; to provide bulk non-volatile storage; and to support atomic transactions with stable storage mechanisms.

The traditional functions of virtual memory are implemented within Alpha in a straightforward fashion. Page replacement is performed according to the system-specified policy, entire objects are swapped out if they have not been accessed in a period of time, and the templates that define object types are stored in a non-volatile part of secondary storage and are accessed when creating object instances. The object invocation mechanism is used within the kernel to obtain access to the structures maintained on the secondary storage devices. This provides all the same features provided to the kernel's client, but at a low level within the kernel (e.g., location-independent access and replication). The use of the operation invocation facility within the kernel also allows a wide range of choices to be made regarding the amount and location of secondary storage devices in the system, as well as the ability to perform swapping and paging to remote secondary storage devices.

This kernel does not provide a file system, but rather that functionality is subsumed by objects in the

Alpha programming model. Objects may have any or all of the properties associated with files (e.g., dynamic creation and deletion, permanence, lifetimes unrelated to their creators, and different access methods). This approach simplifies the system by eliminating the need to introduce another programming abstraction.

In keeping with the policy/mechanism separation approach used in this kernel, an effort was made to decompose the functionality typically associated with secondary storage into orthogonal components representing the (independent) properties desired. In Alpha, these are the permanence of stored entities across machine (and device) failures, and atomicity of the updates applied to stored entities with respect to the visibility of changes across machine failures. With such mechanisms it is possible to construct the following types of secondary storage — *extended storage*, which provides volatile bulk storage, constructed using slower, lower cost storage technology; *permanent storage*, which is similar to extended storage, but is non-volatile and therefore incurs a greater access cost; and *stable storage*, which makes updates to secondary storage appear atomic with respect to system failures, at a proportionally high cost of use. All objects have secondary storage representations (for paging, swapping, migration, resiliency, etc.), and these representations are maintained in different types of secondary storage based on attributes of the object. Objects are given secondary storage attributes when they are created, and these attributes may be changed in the course of the objects' lifetimes.

1.3.5. Issues Addressed

The following provides a brief overview of how each of the major issues associated with distributed real-time control systems is addressed by the Alpha kernel mechanisms developed in this work.

1.3.5.1. Distribution

This research explicitly deals with the effects of distribution by means of a technique for efficiently providing reliable, physical-location-transparent communication at a low level within the kernel. The operation invocation facility is the primary kernel service upon which all else is built. Thus, by making the invocation of operations on objects reliable and location-transparent, the primary effects of physical distribution are abstracted away. Furthermore, by having all objects (and even mechanisms within the kernel) use the invocation mechanism, it is possible to enforce a uniform access method to all system resources, regardless of their actual location within the system. System resources can therefore be managed and accessed uniformly regardless of their physical location.

1.3.5.2. Reliability

For the purposes of this work, a commonly used distributed system failure model was adopted [Anderson 81]: the types of failures considered here are the failures of both hardware and software components, in both the system and application domains, including both *transient* and *hard and clean* failures.

The reliability techniques employed in Alpha are supported primarily by kernel mechanisms that provide a client interface at which failures in the underlying system are abstracted into a range of well-defined, predictable behaviors. In particular, the following reliability issues are addressed:

- **consistent behavior of actions** — provided by mechanisms that support (independently) the attributes associated with atomic transactions (i.e., atomicity, permanence, and serializability).

- **availability of services** — provided by mechanisms that allow objects to be replicated and manage the different types of interactions defined on those replicas.

- **graceful degradation** — provided by mechanisms that use an ordering function (currently based on the timeliness constraints and relative importance) associated with all requests for services, in order to sacrifice lower-valued requests in favor of higher-valued ones when resource allocation conflicts arise.

- **fault containment** — provided by mechanisms that place each object in a separate (hardware-enforced) address space, and by separating software components into private system-enforced protection domains, with all interactions restricted to those explicitly allowed by the capability mechanism.

While the Alpha kernel provides a set of mechanisms to support these objectives, its reliability mechanisms are not intended to form a complete facility. The kernel is intended as a framework within which policy issues relating to these reliability techniques can be explored. The areas of atomic transactions and replication are the topics of two separate thesis research projects [Clark 87, Shipman 87], that will explore the policy decisions associated with each area. The mechanisms provided in Alpha for atomic transactions and replication are initial versions of the more complete sets of mechanisms being developed in these on-going projects.

1.3.5.3. Timeliness

Because timeliness is such an important factor in this research, much attention has been given to considerations of time in the specification, design, and implementation of the mechanisms that make up the Alpha kernel. It is not reasonable to characterize computer systems simply as being real-time or not, but rather there is a spectrum, into which all computer systems fit, that defines the degree to

which the system can meet the demands of various real-time applications. The aspects of operating systems that increase their suitability for real-time applications are not manifest merely in the inclusion or exclusion of specific functions, nor does the inclusion or exclusion of any functionality, in itself, make an operating system unsuitable for real-time use. However, the aspect of an operating system design that is most involved in meeting the needs of real-time applications is the manner in which contention for system resources is resolved — real-time operating systems should take into account timeliness constraints when resolving contention for resources. All resource management decisions should be based on the time constraints of the entity making the requests and when contention occurs, resources should be allocated in a manner dictated by the system's overload handling policies in order to maximize usefulness to the application. The programming model defined in Alpha, and the kernel mechanisms that support it, were designed to deal in this way with the problems of time-driven system resource management.

The kernel's programming abstractions aid in the overall objective of global, dynamic, time-driven resolution of contention for system resources. The thread abstraction embodies a distinct run-time manifestation of logical computations which is more appropriate for distributed systems than the conventional process abstraction. This provides a direct means for associating the timeliness requirements that clients specify for their computations with specific location-transparent run-time entities that the kernel manages. In this manner, global importance and urgency characteristics are propagated throughout the system along with threads, for use in resolving contention for system resources (e.g., processor cycles, communication bandwidth, memory space, or secondary storage) according to a client-defined policy. Also, each node in Alpha has hardware support for making and carrying out the kernel's time-driven resource management decisions.

Alpha can thus be employed in a conventional manner that guarantees real-time deadlines can be met for low-level static applications, but it also has significantly greater capabilities for more general real-time command and control applications. The design and implementation of the Alpha kernel's mechanisms provide further support of real-time applications. For example, the synchronization primitives are designed to consider timeliness constraints in their function. Furthermore, the kernel is designed so that the execution of threads can be preempted while executing within the kernel. This is as opposed to some operating systems (e.g., UNIX) that disable preemption within the kernel, thus reducing the amount of synchronization required to maintain the consistency of internal system data structures. An application may spend 40%-50% of its time executing within the operating system, and so a system's responsiveness to the dynamics of real-time applications suffers if preemption is not permitted while control is within the kernel. In addition, the Alpha kernel mechanisms were

designed and implemented so as to ensure that they will not require highly variable or unbounded amounts of time to complete their functions. This is meant to enchance the predictability of behavior, as is frequently desired of real-time systems.

1.3.5.4. Modularity

The modularity requirements are addressed within the Alpha kernel in two ways — modularity for the kernel's clients is supported through the use the object-oriented programming model supported by the kernel, and modularity within the kernel itself is provided by a policy/mechanism separation approach [Brinch Hansen 70].

The kernel provides a simple and uniform interface to its clients that centers around the operation invocation facility. The object programming abstraction supported by the kernel exhibits the same benefits associated with object-oriented programming abstractions in general, among which are information hiding, increased modularity, enhanced uniformity and simplicity of the programming interface, and reduced life-cycle costs [Bayer 79, Cox 86].

The concept of policy/mechanism separation has been shown to be valuable in the design of modular operating system facilities [Brinch Hansen 71, Levin 75, Habermann 76]. Briefly, a *policy* is defined as a specification of the manner in which a set of resources are managed, and a *mechanism* is defined as the means by which policies are implemented [Brinch Hansen 70]. Policy/mechanism separation is a structuring methodology that involves the segregation of entities that dictate resource management strategies from entities that implement the low-level tactics of resource management.

The Alpha kernel is implemented as a collection of kernel-level mechanisms from which policy decisions were carefully excluded. Each major logical function in the kernel is manifest in an individual mechanism, and a great effort was made to ensure a proper separation of concerns among these mechanisms. If the mechanisms are in fact pure (i.e., devoid of policy decisions) and complete, then it is possible to use them in implementing a wide range of the system- and application-level facilities. Modularity is achieved through the separation of functions into mechanisms; implementation changes are restricted to individual mechanisms, and changes in system policy do not require changes in the *functionality* of mechanisms, just changes in the *use* of mechanisms.

1.3.6. Issues Not Addressed

There are a number of issues which have not (yet) been addressed in the Alpha kernel: some were omitted in order to limit the scope of the initial effort to the aspects of this problem that were considered most interesting or important; others were omitted in order to restrict the scope of the initial effort to a manageable level. While these issues were initially not considered in this research, work is currently underway in some of these areas, and plans have been made to address others of them in future work.

Examples of specific issues that are currently not considered in this research include problems related to: heterogeneous application processors, communication subnetworks other than Ethernet (e.g., token-passing buses and rings), specific performance goals, UNIX compatibility, languages, internetwork communication protocols, and security.

Because Alpha is a kernel, it deliberately does not include a number of higher-level functions normally associated with operating systems, such as initial task loading and placement, and reconfiguration. Although the kernel does provide the necessary support for such functions, these functions will be implemented as higher-level operating system policies on top of the kernel mechanisms.

The kernel facilities described here do not necessarily represent the best possible choices for achieving the desired functionality, but rather each facility reflects a collection of global system tradeoffs necessary for its integration. The Alpha kernel, as described here, is not intended to be a production-quality, released and supported facility, for a user community of application programmers. Rather, it is intended as a research testbed whose primary clients will be a small collection of highly qualified system programmers, performing experiments with operating system concepts and techniques for modular, reliable decentralized real-time control systems. The transition of Alpha and its technology from the academic research domain to larger and more application-oriented system contexts is a major activity already underway both inside the Archons project and outside (e.g., in industry and government organizations).

1.4. Status

This book constitutes a snapshot of the first significant milestone in an on-going operating system research effort. The design and initial implementation of the Alpha kernel took place in 1986. The first version was for processing nodes based on the Sun Microsystems version 1.5 processing element, and a second iteration was for nodes based on the version 2.0 processor.

The Alpha kernel is in its initial stages and there remains a great deal of work yet to be done. The success of the effort cannot be accurately assessed until all of the specified functionality is in place, realistic application programs are executed on top of the kernel, the overall system performance is measured and analyzed, and optimizations are applied to the implementation of critical mechanisms.

The implementation of the mechanisms that support the kernel's basic abstractions is complete — i.e., mechanisms that provide objects, threads, and operation invocation. Mechanisms that provide the bulk of Alpha's major facilities also have been finished — i.e., network communications, thread scheduling, and virtual memory. Many of the other kernel abstractions such as capabilities, kernel-defined objects, object and thread manager objects, along with the semaphore and lock objects have been completed as well. These mechanisms comprise approximately 30,000 lines of (commented) C and 1,000 lines of assembly code, which compile to about 64K bytes of object code.

A number of tools have been created in support of the Alpha kernel. These include: a simple pre-processor for providing the object language extensions to C; kernel building, linking, and loading utilities; a program for the specification of distributed program configurations; a utility for monitoring the transfer of packets on the communications subnetwork; and software for processing the data generated by Alpha's integrated performance monitoring facility.

Work is currently underway on the implementation of the remainder of Alpha's components including: the secondary storage subsystem, the facility for nested, serializable atomic transactions, and the initial mechanisms for managing the replication of objects. In addition, work is being done to develop a virtual machine substrate which allows the Alpha kernel to execute on standard (i.e., single processor, with no adjunct processors) Sun Microsystems workstations.

Further extensions of the Alpha kernel are scheduled, and a full-function decentralized operating system will be created with Alpha as its foundation — delivery of this will take place at the end of 1988. A commercial-grade version of the Alpha kernel is being developed by Kendall Square Research Corporation for the Sun Microsystems Sun-3 hardware, their own multiprocessor hardware, and other machines as circumstances call for.

Application software is now being developed to demonstrate the behavior of the kernel in a simulated real-time command and control environment. These experimental programs will serve both as realistic workload generators for stimulating the kernel's mechanisms, and to assist in illustrating desired behavioral characteristics of the kernel. This application/simulation effort is being performed in conjunction with General Dynamics' Fort Worth Division (where Alpha is being run on the same hardware).

Enough is now understood about the needs and bottlenecks of decentralized operating systems in general, and of Alpha in particular, that research can proceed on further architectural enhancements to accelerate the performance of such systems. The design of both accessory (e.g., for Sun-3 processors) and indigenous (e.g., for Kendall Square Research machines) hardware is now being pursued.

Performance is quite important in any real-time command and control system, and the mechanisms that make up the Alpha kernel reflect an awareness of real-time responsiveness requirements. Despite the considerable amount of effort that was directed towards performance concerns, it was recognized that performance is one of several important attributes that must be traded off against each other in good system design. In the design of Alpha, neither the functionality of the system, nor any of the other attributes (e.g., modularity or reliability) was sacrificed for performance through premature optimization. Only when the desired functionality is in place and the system's behavior is analyzed under realistic loading conditions will any appropriate performance optimization be done.

To assist in the process of system analysis, a performance monitoring utility is integrated with the kernel. Preliminary measurements, such as operation invocation and thread scheduling times, are already commensurate with those for more conventional (and less functional) kernels on comparable hardware (e.g., [Cheriton 84a] and [Burke 83]), indicating promise for both the design and implementation of Alpha.

1.5. Outline of the Book

Chapter 2 describes the programming abstractions created for the Alpha kernel to meet the requirements set forth in this Chapter. Chapter 3 defines the client interface provided by the kernel in support of the given programming abstractions. Chapter 4 provides a description of the functional design of the kernel, while Chapter 5 provides a more detailed description of the kernel mechanisms' design. The hardware on which the kernel was constructed, and the implications of this hardware on the design and implementation of the kernel is described in Chapter 6. This is followed by a

comparison of Alpha with other relevant operating system efforts in Chapter 7. Appendix A contains a description of the language extensions used in writing the initial objects for the Alpha kernel.

- UNIX is a trademark of AT&T Bell Laboratories.
- Accent is a trademark of Perq Corporation.
- Ada is a trademark of the United States Department of Defense.
- Ethernet is a trademark of Xerox Corporation.

2
Programming Abstractions

The Alpha kernel is intended to provide its clients with a set of simple and uniform programming abstractions based on the notion of objects [Bayer 79]. Special attention is given to the task of providing client-level uniformity despite physical distribution and the constraints imposed by the system hardware.

The programming paradigm supported by this kernel is known as *object-oriented programming* [Goldberg 83, Cox 86], and the primary abstractions supported by the kernel are: *objects*, *invocations*, and *threads*. In the Alpha kernel, *objects* adhere to the common definition of abstract data types and interact with other objects via the *invocation* of operations on them. *Threads* are defined to be the manifestations of control activity (i.e., the units of concurrent computation and scheduling) within the kernel. The Alpha thread abstraction is in many ways comparable to the process abstraction found in many conventional systems. Unlike conventional processes however, threads move among objects via invocations, and do so without regard for the physical node boundaries of the system. Furthermore, the kernel's object abstraction extends to all system services and devices. Alpha encapsulates all of the system's physical resources, and provides them to the system's clients through a standard object interface.

Figure 2-1 is a snapshot of an example application running on Alpha, showing an application consisting of three separate threads in the process of moving through four different objects. Note that the boundaries presented by physical nodes do not appear in this logical view, and both $\mathbf{Thread_A}$ and $\mathbf{Thread_B}$ are simultaneously active in $\mathbf{Object_3}$.

This variety of object-orientation was chosen for the Alpha kernel because of the belief that such an approach would be well suited to the type of reliability techniques that have been developed for real-time command and control applications [Sha 85a]. The chosen object model provides a simplified (or constrained) control structure among software components, as compared to a more general process and message-based system model, and restricts the accessibility of the encapsulated

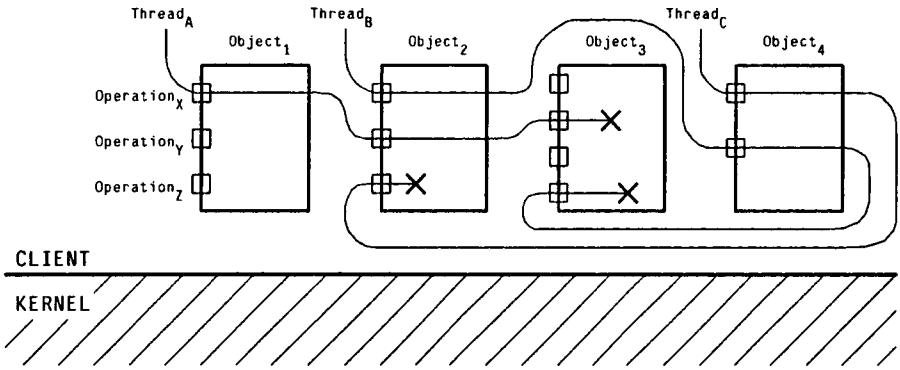

Figure 2-1: Alpha Kernel Example

data items. Among other things, this serves to simplify the task of tracking the operations performed on objects that is required in the implementation of atomic transactions. The fact that the object model centralizes all access to encapsulated data reduces the complexity involved in structuring operations so as to maximize the concurrency that can be obtained from objects (both within and outside of atomic transactions).

Objects most naturally interact through a form of communication characterized by send/wait interprocess communication. Objects in Alpha therefore use a type of Remote Procedure Call (RPC) for the invocation of operations on objects. The RPC style of communication tends to be both simpler and more commonly understood than more general forms of communication (e.g., asynchronous message-passing) [Nelson 81]. The use of a synchronous form of communication in Alpha does not pose the types of limitations that are typically associated with synchronous communications in process/message-based systems [Liskov 85], due to the nature of the object and thread abstractions. Furthermore, there was initially reason to believe that this type of system could be implemented in a way that is suitably efficient to make possible the construction of meaningful applications programs.

A wide range of benefits are claimed for the object model of programming, including increased modularity, the separation of specification from implementation, and the increased reusability of software components. These claims are accepted as true in Alpha, and a general discussion of the merits of object-oriented programming can be found in [Cox 86].

The Alpha kernel is based on a small set of basic mechanisms, similar to the those in Accent [Rashid 81]. The Accent kernel is based on the process and interprocess communication abstractions, while

the Alpha kernel is based on the abstractions of objects, the invocation of operations on objects, and threads. As in Accent, where system calls are performed by sending messages to processes, all kernel services in Alpha are provided by the invocation of operations on objects.

2.1. Basic Abstractions

Alpha implements an interface on top of the system hardware that provides the kernel abstractions of objects, operation invocation, and threads to the client.

2.1.1. Objects

At a high level of abstraction, the Alpha object model is similar to most others (e.g., [Allchin 83], [Almes 85], or [Schantz 85]) in which objects adhere to the common definition of simple abstract data types. In Alpha an object is an entity defined by the data that it encapsulates and a set of operations which are provided to manipulate that data. An object, furthermore, can only be accessed via the operations it exports as a part of its interface (i.e., the object's operation entry points). Additionally, the operations defined on an object specify the number and types of parameters that are to be passed into and out of the object when the operations are invoked.

2.1.1.1. Fundamental Characteristics

The Alpha kernel does not take the object model as far as more "pure" object-oriented systems, and objects are intended to be medium to large in size — i.e., much larger than integers, larger than simple procedures, but smaller than entire programs (e.g., between 100 and 10,000 lines of code). Unlike systems such as Smalltalk [Goldberg 83], not all functions in Alpha are implemented as objects; objects in Alpha may be composed of subroutines. The Alpha kernel reflects the belief that the appropriate granularity of objects is related to the cost of inter-object communication, and that the cost of interprocessor communication in a distributed computer system suggests the use of medium- to large-scale objects. Thus, Alpha supports medium-sized objects that are implemented with (more or less) standard process-style techniques, on (more or less) standard hardware. The Alpha kernel is not constructed on a capability-based addressing architecture, but rather on traditional process-based hardware, unlike such systems as CAP [Wilkes 79] and the Intel 432/iMAX [Kahn 81].

A simplified example of an object in Alpha is illustrated in Figure 2-2. In this example, the object is a prototypical queue object named ***Queue***, with three client-defined operations: **INITIALIZE**, **INSERT**, and **REMOVE**. This queue object includes the data that make up the elements of the queue, the code,

which implements the operations, and the other data, which comprise the internal implementation of the object (e.g., storage for the queued elements, various utility subroutines, pointers used to keep track of the entries within the object, or the data required for internal synchronization). Figure 2-2 reveals the internal structure of the object — only the entry points defined at the interface are visible to the objects that make use of it.

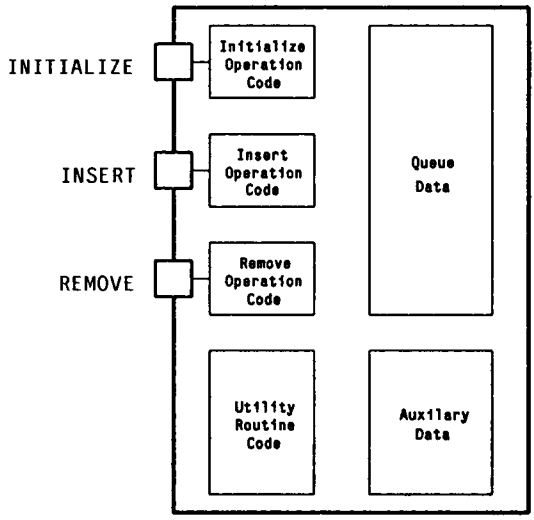

Figure 2-2: Example Queue Object

In addition to the general characteristics of objects defined above, objects in Alpha have the following characteristics. Each instance of an object has a globally unique identifier that can be used to invoke operations on it. The kernel supports a uniform, flat universe of objects; all objects in the system, whether they are a part of the operating system or the application, are undistinguished by the kernel. Objects can be dynamically created and destroyed in the course of the system's execution, can migrate among nodes dynamically, and can have their state frozen and unfrozen (to assist in system debugging). The kernel provides object management services through a kernel-provided object, and accomplishes the dynamic control of objects (i.e., the creation and deletion of different types of objects) through the invocation of operations on this fundamental, system-defined object.

In Alpha, objects are passive entities — i.e., there is no activity in an object until an operation has been invoked on it. Upon operation invocation, an object becomes active — i.e., executes the code associated with the invoked operation (which may, in turn, involve the invocation of operations on

other objects). Once the operations invoked on an object are complete, the object once again becomes inactive and awaits further invocations.

Objects in Alpha represent non-intersecting protection domains [Lampson 69]. Each object exists in a private, hardware-supported virtual address space (or context) that provides a measure of protection from interference caused by other objects. All interactions between objects are performed and controlled by the kernel-provided operation invocation mechanism. A simple capability mechanism [Dennis 66] is provided, with which a range of different access control policies can be implemented. Access to objects is controlled by the use of capabilities that grant the possessor the right to invoke a given operation on a particular object. Since all interaction among objects is performed by the kernel's invocation mechanism, the capability mechanism provides a global, uniform means of controlling access to the objects within the system.

In the Alpha kernel, all objects are instances of various object types. An object *type* is a template that defines the structure and initial value of the data an object should have and the operations associated with the object. Object *instances* are created from a given type, and are the individual run-time manifestations of objects in Alpha. An object is instantiated by invoking a operation on the kernel-provided object management object, with creation-time attributes passed as parameters of the invocation (e.g., the type of the desired object or the object's secondary storage characteristics). Similarly, an instance of an object is destroyed by invoking an operation on the kernel-provided object management object, and passing with it an input parameter that identifies the object to be destroyed.

In addition to the client-specified operations on objects in Alpha provided by the type specification, there is a set of special, system-provided operations defined on all objects. Each object has such a set of *standard operations* defined on it, that are used to manipulate the object's representation in an object-oriented manner. Examples of such kernel-defined operations include operations to suspend and continue the execution of threads within an object, an operation to write the current state of an object out to its secondary storage image, an operation to move an object to another node, and operations to commit and abort the operations performed on an object by threads within atomic transactions. These operations are useful for such purposes as the debugging objects, dynamic reconfiguration, performing atomic transactions, and modifying an object's secondary storage representation.

Unlike client-defined operations, some standard operations are not intended to be invoked directly by client objects, but rather by the kernel itself. For example, exceptions or other kernel-generated

events can be signaled asynchronously (and in a potentially unsolicited manner) to objects by way of the kernel invoking a standard operation on an object. It is also the case that clients may provide custom versions of an object's standard operations. In such cases, the client-provided operations take precedence over the kernel-defined (default) operations. An example of an instance where this feature proves useful is in compound transactions (see Subsection 2.2.3). In this case, the client may increase the performance of applications which perform atomic transactions on objects, by providing a specialized set of operations to replace the standard operations for atomic transactions. The client-defined replacement operations take advantage of an understanding of the semantics of the object's operations to provide the desired effects of atomic transactions more efficiently than would be done automatically by the kernel.

2.1.1.2. Optional Attributes

The kernel provides mechanisms to allow clients to construct objects with a range of differing characteristics that allow differing tradeoffs to be made among such attributes as performance and reliability. A typical tradeoff makes an object appear more reliable (in the sense that there a greater probability that the effects of operations invoked on it will persist across failures), at the cost of performance (in the sense that the operations will take longer to complete). When an instance of an object is created, certain optional attributes of the object may be specified. In addition to being able to specify these object attributes when an object is created, the standard operations of an object provide a means by which an object's attributes to be modified during the course of an object's execution. Among these attributes are a pair of choices that are related to an object's secondary storage representation. These object attributes are known as:

- **permanent** — the state of the object persists across node failures. The last consistent secondary storage image of an object is used to reconstitute the object on restart. This is as opposed to the default **transient** attribute, where the current state of the object is lost when a node failure occurs.

- **atomically updated** — changes to the secondary storage image of an object are made atomically with respect to both normal system behavior and failures. Such an object exhibits the property of atomically changing from one consistent state to another (where consistency is defined by the client on a per-object basis), and at no time can the object be observed in some intermediate state. The default attribute of all objects is that they are not atomically updated.

Another optional attribute of objects is one that permits some objects to seem reliable, in the sense that they can complete the requested operation in the face of (variously) communication failures, node failures, and system software failures. This attribute is known as *availability* (i.e., the probability that the object will be available to provide a desired function). The kernel provides mechanisms to

support the increased availability of objects through the use of a technique known as *replication*. When an object is created a specific replication policy is specified, as well as the degree to which the object should be replicated. Replicated objects provide a range of consistency and availability at a range of costs in terms of invocation latency.

2.1.2. Operation Invocation

Objects may invoke operations on other objects in the course of their execution. The invocation of an operation transfers execution from the *invoking* object to the *invoked* object, and the only data shared between the invoking object to the invoked object is passed in the parameters of the invocation. Each invocation is concluded by a reply, with which the invoked object may return to the invoking object a similar set of parameters.

2.1.2.1. Basic Features

Operation invocation is the means by which all objects interact, and is the global, uniform interface to all client-defined objects, system services, and physical devices in the system. The invocations of operations on objects in Alpha may be nested, and recursive invocation of operations on objects is also permitted. Furthermore, in Alpha invocations are made independent of the physical location of the source and destination objects. While information about the target object's physical location is not necessary to invoke operations on objects, it is made available to those functions which require it (e.g., task placement, or reconfiguration). The physical-location-transparency made possible by the operation invocation facility facilitates the simple and efficient migration of objects from one node in the system to another.

All operation invocations require an identifier for the destination object, an identifier for the operation to be performed, and zero or more parameters (that may include the identifiers of other objects). Similarly, one or more parameters may be returned from an object following an invocation. The one parameter that is always returned from an invocation is a status indication of whether the invocation has succeeded or failed; additional parameters can be returned to indicate the cause believed to be responsible for the invocation's failure. Invocation parameters are passed into the invoked object's domain on invocation and when the invocation is complete, parameters are passed back to the invoking object's domain. All invocation (and reply) parameters are passed by value, and two different types of parameters can be passed — *variables* and *capabilities* (which are system-protected object descriptors and will be explained in more detail in Section 2.2.1).

The fact that invocation is the only manner in which objects can (directly) interact has many important benefits. Invocations provide the kernel with complete visibility of actions among objects. The kernel can track activities of threads — a feature that is useful in the support of atomic transactions, monitoring, and debugging. No alternative channels of communication exist, so the kernel can create a more accurate model of the interactions among the objects it supports, and can follow the execution of computations through successive invocations of objects — all of which contributes to the system's ability to manage system resources more effectively. The invocation mechanism serves as the single point through which all data is passed among objects — this provides an obvious point where translations can be performed on exchanged data to accommodate differences in machine-dependent data representations.

The syntax of invocations, the manner in which parameters are passed and returned, how objects are named, the way capabilities appear to the client, and how capabilities are distributed among objects depends largely on the choice of the language interface provided the client of the Alpha kernel. It is considered the responsibility of this language to manage the initial access restrictions (e.g., by the distribution of well-known capabilities), and to perform whatever degree of compile-time restriction enforcement that is desired (e.g., invocation parameter type-checking). The kernel provides a powerful set of mechanisms to support such a language interface, but it is the responsibility of the language designer to decide exactly how these functions are to be provided to the client. As this research does not enjoy the benefit of an accompanying compiler effort, object programming is done in the C language with simple extensions provided by a pre-processor. Figure 2-3 shows a portion of an example object written for use in the Alpha kernel, and Appendix A provides a description of the language extensions.

2.1.2.2. Flow of Control

At the client interface, the invocation of operations serves as both the means of interaction among objects and the mechanization of the interface between the object and the system. These are clearly at two different levels of abstraction; the interaction between the invoking object and the system is at the lower level, while the interaction between the invoker and the invoked entity is at the higher level. At the lower level of abstraction, the control behavior is by nature synchronous — when the invocation is made the invoking entity's logical progress is suspended while the system (and possibly the invoked entity) performs some work on the invoker's behalf. Alternatively, the type of control behavior found at the higher level of abstraction can generally be categorized as either *synchronous* or *asynchronous*. Synchronous behavior at this level is represented by remote procedure calls, where the invoking process is suspended until the invoked process completes the operation specified in the

```
/*
 * Object Declaration
 */

OBJECT(Queue1)


/*
 * Declarations
 */

#include TRUE  1
#include FALSE 0

#include Q_SIZE        100

static boolean  qfull,
                qempty,
                init = FALSE;
static char     queue[Q_SIZE];
static int      qhead,
                qtail;


OPERATION  Initialize()
/*
 * Initialize the queue.
 * Set the initialization flag and return SUCCESS.
 */
{
    /* initialize the queue */
    qhead = qtail = 0;
    qfull = FALSE;
    qempty = init = TRUE;

    /* return a success indication */
    RETURN SUCCESS;
};


OPERATION  Insert(IN char:chr)
/*
 * Take a character and insert it in the queue if it is not full.
 * If the queue has not been initialized, return FAILURE.
 * If the queue is full return FAILURE, else return SUCCESS.
 */
{
    if (init != TRUE)
        RETURN FAILURE(QNOTINIT);

    if (qfull == TRUE)
        RETURN FAILURE(QFULL);

    /* insert a character in the queue */
    queue[qhead++] = chr;
    qhead %= Q_SIZE;

    /* check if the queue is now full */
    if (qhead == qtail)
        qfull = TRUE;

    /* indicate that the queue is not empty */
    qempty = FALSE;

    /* return a success indication */
    RETURN SUCCESS;
};
```

Figure 2-3: Example Source Code for an Alpha Object

```
OPERATION  Remove(OUT char:chr)
/*
 * Get a character from the head of the queue and return SUCCESS
 *  if the queue was not empty and return FAILURE otherwise.
 */
{
    if (init != TRUE)
        RETURN FAILURE(QNOTINIT);

    if (qempty == TRUE)
        RETURN FAILURE(QEMPTY);

    /* remove a character from the queue */
    chr = queue[qtail++];
    qtail %= Q_SIZE;

    /* check if the queue is now empty */
    if (qhead == qtail)
        qempty = TRUE;

    /* indicate that the queue is not full */
    qfull = FALSE;

    /* return a success indication */
    RETURN SUCCESS;
};
```

Figure 2-3, continued

invocation. At the same level, asynchronous behavior is represented by message-passing systems where the process which sends a message is suspended only to the point where the system can register the message transmission request and then the invoker's progress continues, independent of the destination process.

The lower level of communication must be considered separately from the higher level; the lower level is by nature synchronous, while the higher level can be either synchronous or asynchronous. Given that the lower-level invocation semantics are by nature synchronous, the only interesting choices have to do with the meaning that is associated with the resumption of execution of the invoking entity (i.e., the return of the invocation). Examples of possible meanings associated with the return of invocations include: *the invocation request has been noted, the invocation request is in the process of being serviced, the invocation message has been delivered to the destination, the invocation message has been acknowledged by the destination object, the invocation has been performed by the destination object and has responded,* etc. The higher-level aspect of communication semantics is most commonly thought to be a binary choice between synchronous and asynchronous communication services. In Alpha however, we note the continuum of choices available for an invocation mechanism, and choose the semantics most appropriate for our system. Therefore, the object invoca-

tion mechanism in Alpha is of the extreme synchronous type of high-level, end-to-end communication.

The Alpha kernel does not provide a message-style communication facility and there are currently no plans for adding one. This is not a limitation since it is believed that the Alpha programming model is highly expressive and permits such a high degree of concurrency that an asynchronous message-passing facility is not called for. Furthermore, should asynchronous message-passing be desired, it is possible to construct a message-passing style of communication facility with the existing system. For example, a ***Port*** object might be defined and instances of this type of object may be created. Other objects could invoke a SEND operation on the port object that queues a (client-defined) *message* on the port. A message could then be received by having an object invoke a RECEIVE operation on the ***Port*** object, that would pass back the message as invocation return parameters. Such a ***Port*** object could be a client- or system-defined object. While it is possible to provide a message-style communication facility in Alpha, this would violate some of the basic assumptions of its object model, and would pose a number of technical challenges (particularly with regard to the manner in which atomic transactions could be supported).

2.1.3. Threads

The *thread* is the unit of computation, concurrency and scheduling in the Alpha kernel. All activity in the kernel is provided by threads; objects are passive and all operations are invoked on objects by threads. When a thread is created, an object and an operation within that object is specified as the initial starting point for the new thread. The initial object in which a thread begins is known as the *root* of a thread (or a *root object*). As a thread moves through the root object (and its descendants) making invocations, the thread enters and exits other objects in a nested fashion, independent of the physical node boundaries in the system. Intuitively, a thread corresponds to the conventional notion of a *process*, except that a thread may cross node boundaries in the course of its execution.

Threads are created and destroyed dynamically by invoking operations on a kernel-provided thread management object. To create a thread, an operation is invoked on the thread management object, the object and operation in which the thread is to begin execution are given as parameters, along with the desired attributes of the thread (such as importance or deadline) and any initial parameters. An identifier for the new thread is returned as a result of a successful thread creation operation. Similarly, to destroy a thread, an operation is invoked on the thread manager object, with the identifier of thread to be destroyed given as a parameter to the invocation.

Operations can be invoked on threads as well as objects, and as with objects, operations are invoked on threads by specifying the thread identifier as the destination, and providing the necessary parameters for the operation. Clients may not specify operations on threads; the only operations defined on threads are the standard, kernel-defined operations. The standard operations defined on threads allow threads to stop and restart the logical progress of the thread.

2.1.3.1. Concurrency

Multiple threads can be active within a single object at any point in time, and multiple simultaneous invocations of operations on an object are possible. Because threads execute in asynchronous concurrency with respect to each other, it is sometimes necessary to synchronize the execution of threads. The kernel does not arbitrarily restrict the execution of threads, but rather, each object is responsible for providing the necessary synchronization among threads executing within it. Synchronization among threads is performed within objects via kernel-provided concurrency control mechanisms (details of these mechanisms will be provided in a Section 2.2.2). By placing the responsibility for synchronization with the object, greater concurrency can be obtained than that achievable by blanket, system-provided synchronization techniques. This is because the programmer of an object can use knowledge of the semantics of the object's operations to maximize concurrency among threads by synchronizing only at those points where it is necessary. While applying such brute-force synchronization techniques as monitors [Hoare 74] to objects would relieve the programmer of the burden of providing synchronization, it would also require the entry of individual threads into objects to be serialized, greatly reducing the potential for concurrency.

In the current implementation of the kernel, threads cannot be forked (however, new threads can be dynamically created). It is possible that later versions of the kernel will support the splitting of threads, but this feature will probably not be introduced unless asynchronous invocation services are added to the kernel. This is because some forms of asynchronous communications result in the divergence of threads (or the dynamic creation of a new thread), which currently is not supported.

2.1.3.2. Exception Behavior

While the forking of threads is not supported, certain types of system failures may have similar results. The failure of nodes or communication links can cause threads to become segmented — this results in a problem known as orphan computations [Nelson 81]. Orphans may occur when the execution of a thread extends across multiple nodes. A thread is considered to have been *broken* when a failure occurs at a node that lies along the path between the node a root object is on and the node where the head of the thread is currently executing. The sections of a broken thread, other than

the one that contains the root object, are called *orphans*. Orphans pose a number of problems in the Alpha kernel. Orphans result in an effect similar to the splitting of threads, which is not permitted in the current implementation of the kernel. Furthermore, orphans continue to consume resources as they execute, yet their execution is disconnected from the computation that a thread represents, and therefore cannot contribute to the successful completion of the desired computation. It is therefore important that orphans be detected and eliminated in timely fashion by the kernel. To this end, the kernel's invocation facility supports the notion of *thread repair*. Thread repair involves the detection of the segmentation of a thread, the abortion of all the thread's segments other than that containing the root, and the restoration of the head to the point on the remaining segment that is farthest away from the root.

2.1.3.3. Desirable Characteristics

Another beneficial characteristic of the thread concept has to do with the fact that the client must deal explicitly with concurrency. Each thread in an application represents a computation performed by an independent point of control. In Alpha, concurrency is obtained through the use of threads, as opposed to the use of dynamically created processes or asynchronous message-passing.

Because threads are the physical manifestations of computations, each instance of the kernel can explicitly track and manage the computations running local to it. Threads provide a means of efficiently and effectively resolving contention for system resources. Furthermore, the kernel can assign global priorities (or relative-value functions) to the computations that are active in the system. Therefore, the thread-based approach provides a number of potential benefits over other, more common computational models.

A common alternative to the thread-based approach taken in Alpha is the process and message-based client/server model. In a process/message system, each object would typically be implemented as a separate process, and the invocation of operations on objects would be performed by sending messages to processes. A drawback of such an approach is that there is no single system entity (such as a thread in Alpha) that represents a unit of activity and computation. An example of how this could be undesirable can be seen in the case of contention for the processor — a condition that is frequently resolved by some form of priority scheme, whereby the highest priority process is always executed first. In common message-based systems, the importance of a computation is lost as messages are sent to different servers as the computation progresses. Each step of a computation is performed at the priority of the server, not the priority of the overall computation. In some message-based systems, the priority of a message can be used to resolve conflicts for communication resources. To

resolve contention for all types of resources, computations would have to retain their priority across all of their steps. To do this would require that server processes assume the priority of each message they receive. Furthermore, the server processes must be preemptable so that when a message with higher priority than the current one arrives, the server must suspend its current work and new work must be begun on behalf of the newly received message. It is (at best) a difficult proposition to so intertwine the scheduling and communication facilities of a system. Thus the thread/object approach taken in Alpha seems a much more promising approach to effectively resolving the contention for resources introduced by distributed real-time tasks.

In the Alpha kernel, a thread represents a client-level computation that has a direct physical manifestation within the kernel, and each thread can be assigned such attributes as *importance* (relative to the other threads in the system), and *urgency* (based on the deadlines which it must meet). These features of threads provide the basis for a means of resolving contention for system resources (e.g., processor cycles, communication bandwidth, buffers, or I/O devices) on the basis of the global characteristics of individual computations. Objects in Alpha take on the characteristics of the threads that are executing in them at any point in time. This is analogous to having processes dynamically alter their priority based on that of the process which is the source of the last request message it has received. In Alpha prioritization can be performed on a per-thread (and therefore a per-computation) basis, as opposed to the per-process basis that is typical in most process/message systems. The direct association of logical computations in Alpha and system entities (and their attributes) helps the system in making effective global resource management decisions.

If a computation is thought of as a thread of control that makes use of the functions provided by various objects in performing its function, then each invocation of an operation on an object can be thought of as delineating the sequence of steps that comprise the computation. Given such a view of computations, there is no logical reason why a computation should be required to interact with a system's scheduling facility between each step. Once the scheduling facility has bound a computation to a processor, it should only be unbound when a scheduling event occurs, and the movement of a thread from one step to another does not constitute a valid scheduling event. Clearly, if the scheduling facility must be involved in each processing step, unnecessary system overhead is incurred and it becomes more difficult to ensure that timeliness guarantees associated with the client-level computation can be met. With an object and thread-based approach, a thread moves among its steps without involving the scheduler, until a scheduling event occurs (e.g., a time quantum is exhausted, a higher priority thread is available, or a operation was invoked that blocks the thread). It should be noted that, should a thread cross a node boundary as a result of an invocation, a scheduling decision must

be made because the thread must contend for the processor with the other threads at the destination node.

Threads represent individual points of control that move among the objects and provide the services required by the computations. Also, multiple threads may be active within an object simultaneously. For these reasons, threads and objects in Alpha do not suffer from the nested monitor problem [Lister 77]. Also, when a thread is blocked within an object, it is only the computation represented by the individual thread whose progress suspended. In a process/message system, each process has but a single point of control, and should a server process block, no other requests may be serviced. For this reason, the progress of all of the server's clients may be affected if a single client causes the server to block.

2.2. Additional Abstractions

This section describes the mechanisms that augment the basic abstractions. An access control mechanism is provided to be used in conjunction with the invocation abstraction, concurrency control mechanisms are provided in addition to the thread abstraction, and the atomic transaction and object replication support mechanisms extend the functionality of the kernel's basic abstractions.

2.2.1. Access Control

The Alpha kernel provides mechanisms that may be used to enforce various protection policies. The intent of the access control mechanisms in Alpha is to provide the system with defensive, not absolute, protection. This approach is taken primarily to limit the scope of this research effort to the issues of more immediate concern. Furthermore, the embedded nature of most real-time command and control computer systems restricts (but does not eliminate) the opportunities for mounting determined attacks on the system. The emphasis in this work is more on providing a reasonable degree of assurance (at a moderate cost) that programming errors will not lead to serious system failures.

The primary means of providing this assurance in Alpha is by way of system-provided and enforced protection domains. The kernel places each object in a separate domain, enforces this separation, and controls all interaction among objects and domains. To provide the desired degree of protection, the access control mechanisms in Alpha ensure that each object can invoke operations on only those objects for which it has explicit permission to do so. Furthermore, by enforcing the separation of object protection domains, the general system objective of fault containment is advanced.

Fault containment involves the attempt to limit the effects of the failure of one component on the other components within a system [Levin 77, Boebert 78]. Ideally, fault containment would guarantee that a failed object could not interfere with the operation of any other object. The Alpha kernel's access control mechanisms are designed, however, only to limit the scope of interactions each object may have, and thereby limit the potential extent of a failed object's damage.

The Alpha kernel's protection goals are met with mechanisms that are consistent with the overall philosophy of the kernel. The access mechanisms are designed to be compatible with the objects supported by the kernel, and provide only that degree of protection that is cost-effective within the context of the given application environment.

2.2.1.1. Techniques

Access control in Alpha is provided by a capability mechanism [Fabry 74]. A capability is an object's local manifestation of a system-protected object identifier, and provide objects with a means of accessing the objects they wish to invoke operations upon. To invoke an operation on an object in Alpha, the invoking object must only possess a capability for the object. The very fact that an object possesses a capability implies that the object has the right to access the object referenced by the capability (subject to any restrictions associated with the individual capability). Since all interactions among objects in Alpha are via invocations and all invocations use system protected names, capabilities provide globally uniform access control. Capabilities are used to uniformly solve both the problems of object addressing and object access control in Alpha.

A capability is analogous to a logical pointer to an object that cannot be forged, nor directly manipulated by objects. Capabilities in Alpha are long-lived, i.e., they exist independent of the lifetime of the objects that create them. Capabilities may be explicitly destroyed by an object that has the rights to do so, or they may be lost when objects are destroyed, otherwise they remain in existence along with the objects that possess them. Capabilities are context independent — they describe the same object regardless of their current domain. The kernel maintains the representations of all of the capabilities in the system and controls all access to the them.

The kernel-maintained representation of a capability consists of: a globally unique object identifier that is used to address an object; a list of operation rights that defines which operations (both client- and system-defined) can be performed on the given object; and a per-operation set of usage restrictions that dictate how the capability can be used by the object that owns it (i.e., *no-copy*, *no-transfer*, *single-use*, and *exclusive-use*). Objects can restrict the rights associated with capabilities, but they

cannot add rights that they do not already have. That is, the kernel provides a *restrict* primitive, but no primitive for rights amplification. When an object is instantiated, the creating object receives a capability for the new object, with all of the associated rights and restrictions.

Each object in the system has its own list of capabilities, defining its current access domain (i.e., the other objects on which the object can invoke operations). Some capabilities are given to an object when it is created; these are called *well-known* capabilities. In addition to its well-known capabilities, an object can acquire new capabilities, all of which can be deleted or passed to other objects — depending on the particular restrictions associated with each individual capability. The kernel's invocation mechanism allows capabilities to be passed as invocation parameters as a standard part of its function. The capabilities to be passed, either into or out of an object, are specified as part of the object's interface in the formal parameter list of the appropriate operation. A capability must have the appropriate rights if it is to be passed in an invocation. Furthermore, by using the *restrict* primitive, objects may place additional restrictions on the use of capabilities when they are passed to other objects.

When created, an object could be provided with no capabilities at all, or with well-known capabilities for all of the objects it may need to interact with. A common scheme for capability distribution is one in which all objects are created with a single well-known capability for an object known as a *name server.* A name server object would typically have operations defined on it to allow objects to associate themselves with service names (e.g., client-defined strings), and to return capabilities in response to requests for an object associated with a given service name. In this way capabilities could be distributed to objects at run-time, and higher-level access control schemes could be applied using the name server (e.g., access control lists).

Each time an object is invoked, all of the capabilities used in the invocation are validated. As part of the invocation procedure, the kernel checks that the capability for the invoked object is valid, and similarly checks the validity of all capabilities passed as parameters.

As with many other systems that use capabilities [Levy 84], in Alpha there are no provisions for the actual revocation of capabilities. In the Alpha kernel, capabilities can only be revoked in a limited sense — capabilities may be passed with the restriction that they may only be used for a limitied time, which constitutes a limited form of (automatic) revocation of passed capabilities.

2.2.1.2. Attributes

It should be noted that these protection mechanisms, like all of the mechanisms in Alpha, are not necessarily intended to be the primitives ultimately used by the application programmer. Exactly how capabilities appear to the object programmer, and how they are passed in invocations, depends on the specifics of the language or operating system that is to be built on the Alpha kernel. These kernel mechanisms provide the kernel's clients with a means of inter-object protection.

It was decided that a full capability system (one in which a capability is required for virtually every bit accessed) is not called for in Alpha, and a more suitable solution would be a higher-granularity type of protection scheme. The access control mechanisms used in Alpha are appropriate to the level of granularity of the objects within Alpha — i.e., the protection is performed on a per-object and operation basis, as opposed to a memory segment basis. This choice of protection mechanisms was motivated by the desire to construct the kernel on traditional hardware, the goal that protection should add only a small amount of overhead to the cost of operation invocation, and the belief that there is little is to be gained (in this context) from fine-granularity protection.

Also, the definition of protection domains used in Alpha affords a greater degree of fault containment than a direct access (or access path) approach taken in other object-oriented systems that use capabilities more extensively than Alpha [Jones 79, Wulf 81]. In Alpha, protection domains are compartmentalized, in that each object can only invoke operations on objects that it has a capability for. Other systems allow objects to use capabilities that they themselves do not have by referring to capabilities indirectly, through a chain of other objects' capabilities (e.g., *paths* in Hydra). While objects may still fail in Alpha, they can only access (and hence interfere with) those objects for which they have capabilities. In this way, the compartmentalization of object protection domains helps to confine the effects of object failures. A compartmentalized approach to capabilities allows programmers to set up firewalls that may detect the aberrant behavior of a failed object and halt the propagation of its effects. However, in a system where paths may be used in place of simple capabilities (i.e., indirect access to objects is allowed via chains of capabilities), if an object fails it can directly manipulate objects outside of its immediate protection domain.

The protection mechanisms provided by Alpha are more similar in this respect to the protection schemes found in some message-passing systems than those of object- and capability-based systems. The capability mechanism used in Alpha is similar to that found in some message-passing systems where protection is provided by controlling communication among processes (e.g., via port control, such as in [Baskett 77] and [Rashid 81]), and is most closely related to that of the (centralized) CAL

system [Lampson 76]. The protection service of Alpha is much less comprehensive and provides a lesser degree of protection at a more modest cost than the protection facilities in such systems as Hydra [Wulf 81], StarOS [Jones 79] and System/38 [Berstis 80].

2.2.2. Concurrency Control

Threads, as they are defined in Alpha, are unconstrained with respect to their execution relative to one another. In particular, multiple threads may execute concurrently within a single object. In order to construct applications that behave correctly and predictably, it is necessary for the kernel to provide mechanisms for controlling the concurrency among threads. The concurrency control mechanisms provided by Alpha allow a client to restrict the concurrent activity of threads where necessary to achieve the desired system behavior, while still meeting the goal of maximizing concurrency of the threads in the system. Another objective of the concurrency control mechanisms in Alpha is in providing support for a range of reliable, modular programming disciplines (e.g., various object-oriented languages or atomic transactions).

Most concurrency control mechanisms are based on the notion of controlling (in the sense of starting and stopping) the logical progress of computations, which translates to the control of thread execution in Alpha. Therefore, controlling the virtual progress of threads is the basis for all concurrency control mechanisms in Alpha, and is the fundamental technique that provides support for higher-level synchronization facilities.

Because the object model constrains all access to data to be from within the object that encapsulates that data, verification and enforcement of various synchronization conventions is greatly simplified. In much the same way that monitors are an improvement over the use of generalized critical sections, objects centralize the location of the code that shares access to particular pieces of data and therefore also centralizes the locations where concurrency control is required. Furthermore, objects make possible the use of semantic information concerning the operations on the constituent data to obtain greater application-level concurrency (both within and outside of atomic transactions) than is possible through more standard techniques (e.g., processes). For example, with objects it becomes more reasonable to consider such measures as enforcing orderings on the individual steps of the operations defined on an object in order to obtain increased concurrency [McKendry 84b].

2.2.2.1. Thread Mutual Exclusion

The first (and most basic) synchronization mechanism provided by the Alpha kernel can provide functionality similar to that of *critical sections*. This concept restricts concurrent access to a given region of code to a maximum of **N** threads within an object (of course the special case where **N** = 1 allows the mutual exclusion of threads to be enforced over the critical section). Critical sections are useful for ensuring that at any one time, a maximum of some given number of threads can be executing a section of code within an object. Once the maximum number of threads have entered and are executing within a critical section, all other threads that attempt to enter this part of an object are made to wait (i.e., they are *blocked*) until one (or more) of the threads leaves the critical section. The mechanism commonly used to implement this abstraction is known as a *counting semaphore*, which Alpha supports in the form of kernel-provided ***Semaphore*** objects and a semaphore management object.

Semaphore objects are allocated and deallocated by invoking operations on semaphore management objects. The ***Semaphore*** objects have operations corresponding to the traditional P and V operations of semaphores defined upon them. Additionally, the ***Semaphore*** objects have a non-blocking (or conditional) `P` operation that returns a success or failure response instead of blocking the thread. Individual ***Semaphore*** objects are associated with the object that created them (via an allocation operation on the semaphore management object), and operations can only be performed on ***Semaphore*** objects from within the object that created them.

The semaphore mechanism can be used in conjunction with the objects in Alpha to implement monitor-like structures. For example, when an object is initialized it could create a ***Semaphore*** object with a count initially equal to one. Each operation in the monitor-like object could then begin by invoking a `P` operation on the previously allocated ***Semaphore*** object, and end with a `V` invocation on the same ***Semaphore*** object. This ensures that, like a monitor, each object (that adheres to this discipline) can have exactly one thread active in it at any time. It should be noted, however, that such monitor-like structures quite severely restrict concurrency, and much greater concurrency can be achieved through a more judicious use of concurrency control mechanisms. For example, in many instances, higher concurrency could be achieved if critical sections were used only in those parts of the code where undesirable side-effects may occur due to the concurrent execution of threads within an object. This is merely noting that the granularity of synchronization can be made smaller than the entire object in order to increase concurrency. Also worth noting here is the fact that semantic information concerning the operations being performed by an object must be applied in order to achieve greater concurrency; if nothing is known about an object's operations, only simple, restrictive forms of synchronization can be used.

An example in which the thread mutual exclusion mechanism is used to coordinate the activity of multiple threads is shown in Figure 2-4. This example uses the same ***Queue*** object shown in Figure 2-3, except in this case the fact that multiple threads may be active in the object at any one time is addressed. In this example a semaphore is used to enforce the restrictions that the number of threads active in the REMOVE operation must be less than or equal to the current number of elements in the queue, and the number of threads active in the INSERT operation must be less than or equal to the current number of free entries in the ***Queue*** object.

There are a number of observations that should be made on the nature of this thread mutual exclusion mechanism. It is expected that the language (or operating system) that is to be constructed on top of the Alpha kernel will eliminate the need for the explicit use of this synchronization mechanism by the client. For example, a language might have a block-structured *Critical Section* primitive, a *Monitor* declaration, or some higher-level synchronization primitive. In such languages the allocation and deallocation of ***Semaphore*** objects, as well as the insertion of P and V operation invocations at the appropriate points within objects, would be performed automatically. A client-level synchronization policy could allow some of the synchronization activities to be implicit, even if it results in less than optimal concurrency. An additional note concerns future extensions of the thread mutual exclusion mechanism. In later incarnations of the Alpha kernel, issues of timeliness will receive greater attention and the inclusion of timeouts in synchronization primitives will be considered. Such a change will have to be reflected in the interface to and implementation of the semaphore mechanism.

2.2.2.2. Data Item Locking

In addition to providing mutual exclusion of threads to portions of code within objects, it is also necessary to control thread access to data within objects. This functionality is provided by another concurrency control abstraction known as *locking*. In Alpha, locking is how objects control access to the data they encapsulate. By locking only the data that is being manipulated within an object, greater concurrency is generally obtainable than through enforcing mutual exclusion of threads on regions of code. The potential for concurrency increases because different pieces of data can be manipulated by each thread concurrently executing a particular piece of code, and so multiple threads could be allowed to execute in the same sections of code without interference.

An object needing to synchronize access to its data would allocate a ***Lock*** object, specifying as parameters the data region with which the lock is to be associated. When the specified data is to be accessed, a thread must first successfully acquire the lock. The desired operation is performed on the locked data, and finally, the lock is released. In Alpha, locks are associated with specific items of data

```
/*
 * Object Declaration
 */
OBJECT(Queue2)

/*
 * Declarations
 */
#include TRUE          1
#include FALSE         0
#include Q_SIZE        100

static boolean         init = FALSE;
static char            queue[Q_SIZE];
static int             head, tail;
WELLKNOWN CAPA         SemaphoreManager      SemObj;
CAPA                   Semaphore      FullSlots, EmptySlots;


OPERATION  Initialize()
/*
 * Initialize the Queue. Allocate the semaphores, set the
 *  initialization flag and return 'SUCCESS' to caller.
 * Return 'FAILURE' if allocation of semaphores is unsuccessful.
 */
{
    /* initialize the queue */
    head = tail = 0;
    init = TRUE;

    /* allocate and initialize the 'FullSlots' semaphore */
    INVOKE SemObj.Create(0, FullSlots);

    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(SEMFAIL);

    /* allocate and initialize the 'EmptySlots' semaphore */
    INVOKE SemObj.Create(Q_SIZE, EmptySlots);

    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(SEMFAIL);

    /* return a success indication */
    RETURN SUCCESS;
};


OPERATION  Insert(IN char:chr)
/*
 * Take a character and insert it in the queue if it is not full.
 * If the queue is full, block the thread until space is available and
 *  then insert the character into the queue and return 'SUCCESS'.
 */
{
    if (init != TRUE)
        RETURN FAILURE(QNOTINIT);

    /* get an empty slot */
    INVOKE EmptySlots.P();

    /* insert a character in the queue */
    queue[head++] = chr;
    qhead %= Q_SIZE;

    /* produce a full slot */
    INVOKE FullSlots.V();

    /* return a success indication */
    RETURN SUCCESS;
};
```

Figure 2-4: Example of Thread Coordination

```
OPERATION  Remove(OUT char:chr)
/*
 * Remove a character from the queue, if it is not empty.
 * If the queue has not been initialized, return 'FAILURE'.
 * If the queue is empty, block the thread until a character is available
 *   and then remove a character from the queue and return 'SUCCESS'.
 */
{
    if (init != TRUE)
        RETURN FAILURE(QNOTINIT);

    /* get a full slot */
    INVOKE FullSlots.P();

    /* remove a character from the queue */
    chr = queue[tail++];
    tail %= Q_SIZE;

    /* produce an empty slot */
    INVOKE EmptySlots.V();

    /* return a success indication */
    RETURN SUCCESS;
};
```

Figure 2-4, continued

within an object, and the locking of the data is accomplished by delaying the virtual progress of threads which attempt to access it — i.e., a thread blocks until it is able to lock a data item. If another thread already holds a lock on a data item, other threads will block when attempting to lock the data, and be unblocked when they successfully acquire the lock. Just as with semaphores, the lock abstraction appears to the clients of Alpha as kernel-provided ***Lock*** objects, which are allocated and deallocated by invoking operations on a kernel-provided lock management object. As with ***Semaphore*** objects, only the the object that created a ***Lock*** object can invoke operations on it. The individual ***Lock*** objects provide operations for locking data items, conditionally locking data items, unlocking data items, and modifying locks already held on data items. The lock operation is used by a thread to indicate its desire to access the data specified in the LOCK operation. The conditional lock operation is similar to the LOCK operation except it does not impede the progress of the thread if the lock cannot be granted, but returns a status indication in any event. The UNLOCK operation is used by a thread to indicate that the current manipulation of the previously locked data item is complete. The lock conversion operation is used by threads to modify the type or scope of a currently held lock without releasing it.

In addition to this basic functionality, locks in Alpha accommodate the fact that there are different types of accesses that can be performed on the data within objects. Rather than enforcing mutually exclusive access to data within objects by threads, increased concurrency may be achieved through the use of different types of locks that reflect the type of manipulation that is to be performed on the data. For example, it is frequently the case that multiple simultaneous reads of a data item may take

place without requiring synchronization among the threads reading the data. To express the types of manipulation to be performed on data, and which types of manipulation are compatible, the Alpha kernel introduces the notions of *lock modes* and *lock compatibility tables* [Bayer 79]. A lock mode specifies the kind of operation that a thread attempting to acquire the lock intends to perform on the data associated with the lock. A lock compatibility table specifies which lock modes are compatible with other, currently granted lock modes. A lock is termed *compatible* with another lock (i.e., the locks do not conflict) if the actions defined by the lock mode can be meaningfully performed concurrently. The lock modes defined in Alpha are:

- **Concurrent Read** — allows multiple readers of the data associated with the lock, but only one thread at a time may hold any form of write lock.
- **Concurrent Write** — allows multiple readers and multiple writers of the data associated with the lock.
- **Exclusive Read** — only one thread can have a read lock on the data associated with the lock, and only one writer is allowed.
- **Exclusive Write** — only one thread can have a write lock on the data associated with the lock, and multiple readers are allowed.
- **Exclusive Read/Write** — provides complete mutual exclusion to the data, only one thread can have access to the lock's data.

The compatibility table for locks in the Alpha kernel is shown in Table 2-1.

Requested \ Granted	Concurrent Read	Concurrent Write	Exclusive Read	Exclusive Write	Exclusive Read/Write
Concurrent Read	—	—	NC	—	NC
Concurrent Write	—	—	—	NC	NC
Exclusive Read	NC	—	NC	—	NC
Exclusive Write	—	NC	—	NC	NC
Exclusive Read/Write	NC	NC	NC	NC	NC

— : Compatible	NC : Not Compatible

Table 2-1: Lock Compatibility Table

As is the case with the other synchronization mechanisms in Alpha, the language or operating system that serves as the kernel's client would ideally make the locking of data within objects implicit. However, it is only with the knowledge of the semantics of the operation in question that locking can be done so as to attain maximum concurrency. Most simple attempts at compile-time lock generation result in less than optimal concurrency.

The use of locking mechanisms is usually accompanied by the possibility of deadlock. While the problem of deadlocks is currently not addressed in Alpha, extensions to the lock mechanisms are planned to associate timeouts with individual ***Lock*** objects. This is to aid in recovery from deadlocks, livelocks, and other failure conditions where computations are not making expected progress.

In addition to its role in concurrency control for objects, the locking mechanism plays a central role in the implementation of atomic transactions in the Alpha kernel. Some aspects of the functionality of the lock mechanism in Alpha are specifically provided for the support of atomic transactions. The attribute of serializability typically associated with atomic transactions can be attained through a two-phase locking discipline applied to the use of the locking mechanism [Eswaran 76]. The Alpha kernel employs an optimistic strategy in supporting the atomic update of modifications made by atomic transactions — each lock has a write-ahead log associated with it to allow the changes made to the locked data item to be undone in case an atomic transaction aborts. Details of the manner in which locks are used in atomic transactions will be presented in the following section.

An example of the use of locks is shown in Figure 2-5. This is the same queue example used previously, but now with locking of the ***Queue*** object's queue data and head and tail pointers to ensure that the (potentially **N**) threads that may be executing in the object do not corrupt the queue. By locking the queue pointers, consistency can be guaranteed with a greater degree of concurrency than achievable through simple mutually exclusive access to the object's code. Note that in this case there is no useful activity that the threads may be performing and it is therefore doubtful that the marginal increase in potential concurrency would offset the overhead associated with performing the locking operations. However, more complicated examples could be imagined for which the potential benefits of additional concurrency are much more significant.

```
/*
 * Object Declaration
 */

OBJECT(Queue3)

/*
 * Data Declarations
 */

static int              head, tail;
static char             queue[Q_SIZE];
WELLKNOWN CAPA          Lock    LockObj;
CAPA                    Lock    QueueLock, HeadLock, TailLock;

OPERATION  Initialize()
/*
 * Initialize the queue and return SUCCESS to caller.
 * Allocate lock objects for the queue data structure and
 *  the queue head and tail pointers.
 */
{
    /* initialize the queue */
    head = tail = 0;

    /* allocate the queue lock */
    INVOKE LockObj.Create(QueueLock, queue, Q_SIZE);
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(LOCKFAIL);

    /* allocate the head and tail pointer lock */
    INVOKE LockObj.Create(HeadLock, &head, sizeof(head));
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(LOCKFAIL);

    INVOKE LockObj.Create(TailLock, &tail, sizeof(tail));
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(LOCKFAIL);

    /* return a success indication */
    RETURN SUCCESS;
};


OPERATION  Insert(IN char:chr)
/*
 * Take a character and insert it in the queue if it is not full,
 *  and return a SUCCESS, otherwise return a FAILURE.
 */
{
    /* check if queue is full */
    INVOKE HeadLock.Lock(ConcurrentRead);
    INVOKE TailLock.Lock(ConcurrentRead);
    if (((head + 1) % Q_SIZE) == tail)
        RETURN FAILURE(QFULL);
    INVOKE TailLock.Unlock();

    /* bump head pointer, and roll it around if necessary */
    INVOKE HeadLock.Lock(ExclusiveReadWrite);
    ++head %= Q_SIZE;
    INVOKE HeadLock.Lock(ConcurrentRead);

    /* lock the queue, insert a character into it, and unlock it */
    INVOKE QueueLock.Lock(ExclusiveWrite);
    queue[head] = chr;
    INVOKE QueueLock.Unlock();
    INVOKE HeadLock.Unlock();
};
```

Figure 2-5: Example of Locking Within Objects

```
OPERATION  QOut(OUT char:chr)
/*
 * Remove a character from the queue, if it is not empty,
 *   and return a 'SUCCESS', otherwise return a 'FAILURE'.
 */
{
    /* check if the queue is empty */
    INVOKE TailLock.Lock(ConcurrentRead);
    INVOKE HeadLock.Lock(ConcurrentRead);
    if (head == tail)
        RETURN FAILURE(QFULL);
    INVOKE HeadLock.Unlock();

    /* lock the queue, remove a character from it, and unlock it */
    INVOKE QueueLock.Lock(ExclusiveRead);
    chr = queue[tail];
    INVOKE QueueLock.Unlock();

    /* bump tail pointer, and roll it around if necessary */
    INVOKE TailLock.Lock(ExclusiveReadWrite);
    --tail %= Q_SIZE;
    INVOKE TailLock.Unlock();
};
```

Figure 2-5, continued

2.2.3. Atomic Transactions

Atomic transactions have been shown to be very useful in the construction of reliable applications such as database systems [Eswaran 76, Lampson 81]. Some early reports proposed the use of transactions within distributed operating systems [Lampson 81, Jensen 84], and in recent times the belief that atomic transactions may prove useful within distributed operating systems has become more widely accepted. A number of efforts are currently underway to explore the inclusion of atomic transactions as an operating system service or as language primitives [Popek 81, Liskov 84, McKendry 84a, Almes 85].

Because the primary goal of the Alpha kernel is to support research on the development of distributed operating systems for real-time process control, certain reliability constraints are implied; the kernel itself must function reliably in the face of system component failures, and the kernel must provide the application with mechanisms that will allow the application to function with similar reliability. While other research efforts have explored the notion of atomic transactions, few have attempted to include atomic transaction support within an operating system kernel. Typically these efforts provide atomic transaction facilities on top of existing operating systems. Atomic transaction support was one of the major factors that influenced the design of the Alpha kernel — in fact, it is one of the primary reasons that the object model was chosen as the fundamental programming abstraction.

By integrating the notion of atomic transactions into the design of the kernel, it is anticipated that the

resulting performance of atomic transactions at the application-programming-level should be sufficiently high to allow meaningful experimentation to be performed. Previous attempts at constructing atomic transactions on top of existing operating systems indicate that such an approach can degrade system performance to the point of making it difficult to implement meaningful applications [Spector 84, Almes 85]. Furthermore, the inclusion of atomic transaction mechanisms within the kernel provides reliability support, similar to that provided to the application, for use within the system itself. The same benefits that atomic transactions bring to the construction of application programs is useful in the construction of system software.

The mechanisms currently provided by the Alpha kernel in support of atomic transactions are meant to be representative of the more comprehensive set of mechanisms being developed as a part of a doctoral research project currently in progress [Clark 87].

2.2.3.1. Concepts

The notion of atomic transactions, as they are commonly defined, encompasses a number of different concepts that provide a means of achieving a type of system behavior that is useful in constructing reliable systems. In the context of the Alpha kernel, classical atomic transactions may be viewed as a discipline applied to the use of mechanisms, and a set of programming conventions, that together result in making objects appear to behave in a well-defined manner despite the failure of system components. By enforcing this desired behavior on objects, it has been shown that reliable distributed applications may be constructed [Popek 81, McKendry 84a, Almes 85]. In the Alpha kernel, the commonly accepted definitions of what constitutes well-behaved objects and what types of system failures will be considered are adopted as requirements for this work [Moss 85]. The intent of the Alpha kernel is to support these definitions as baseline requirements and also to provide a vehicle for the refinement of these definitions.

An atomic transaction is traditionally defined as a computation (possibly a part of a broader computation) that performs actions on objects, the effects of which appear to be done atomically with respect to failures and to other transactions, and all transactions appear to execute independently of all others [Lampson 81]. In a simplified view, actions are typically characterized as reads and writes on data that are represented by objects. The major concepts that constitute the classical notion of atomic transactions may be summarized as follows:

- **Atomicity**: This is the property of the *all or nothing*-type behavior of actions — i.e., either all of the individual actions comprising a transaction are successfully performed, or none of them are performed. The effect of atomicity is that (from an external view) the state of the system transitions from one (externally) consistent state to another. In this case,

consistency is defined as some predicate on the data items, known as the *invariant* of the database. While the database itself may be in an (internally) inconsistent state at some point in time, the property of atomicity ensures that this state is not externally visible. Atomicity therefore provides the guarantee that, despite failures of system components, no data object can be observed in a state that does not satisfy the system's invariant conditions.

- **Permanence:** This is a property of objects that ensures the continued existence of the (externally observable) effects of successfully completed atomic transactions, even in the face of system component failures. Once a transaction reaches a successful completion and the effects of its actions are made visible to others, failures in system components will not result in the state of data objects reverting to some previous state.

- **Serializability:** This is a property of atomic transactions having to do with the relative ordering of actions among separate transactions. The individual actions comprising concurrently executing atomic transactions are executed by the system's processors in some partial order known as a *schedule*. A *serial schedule* is one where all of the actions of a particular atomic transaction are executed either before or after all of the actions of any other atomic transactions. A schedule is defined to be *serializable* if its effects are the same as if a serial schedule had been executed. Thus, serializability is the property of atomic transactions that provides the appearance of a non-interleaved execution of individual transactions.

2.2.3.2. Approach

Because of the nature of the intended application domain for the Alpha kernel, the availability of system resources and services is of equal (if not greater) importance to providing for consistent restart after an arbitrary period of unavailability. In a real-time command and control system the quality of information tends to degrade over time. Stored data is therefore not useful during the down period of the node at which it is stored and when the node recovers the stored information may be invalid, even if it was consistent at some earlier time. Thus, the definition of consistency in a real-time system must include a specification of time in addition to the normal system invariants. In these respects, the Alpha kernel differs from many other database-oriented applications that use atomic transactions (e.g., banking systems or airline reservation systems). This is why an atomic transaction facility is a necessary, but not sufficient, part of meeting the system's reliability goals, because atomic transactions provide some, but not all, of the properties of good behavior that are needed in constructing reliable applications.

The atomic transaction mechanisms provided by the Alpha kernel are meant to be general mechanisms for use by the authors of both system and application code. The kernel does not enforce any policy on the use of these mechanisms, nor does the kernel automatically apply atomic transactions to client-defined code — the client chooses when and where atomic transactions are to be used.

The basic form of atomic transactions supported by mechanisms in the Alpha kernel have the additional attribute of *nesting*. This attribute allows atomic transactions to be placed (completely) within other atomic transactions. This is done to provide a finer level of granularity than can be achieved by placing all actions within one large, top-level atomic transaction. Nested transactions also provide a form of modularity in which an object may use atomic transactions to achieve its given level of reliability, and this use of atomic transactions is not visible to invoking objects. In a nested atomic transaction, if a lower-level atomic transaction fails it is reported back to the level that initiated the transaction, where it is decided by the client's code whether the transaction should be retried or whether this level should abort to the next level up.

In addition to providing the basic functionality of atomic transactions as described in the foregoing subsection, the Alpha kernel provides mechanisms to allow further research in the area of modular, high-concurrency transactions (based on related research [Allchin 83, Sha 85a]). One particular area of atomic transaction research that the Alpha kernel is meant to support is the exploration of the notion of *compound transactions* [Sha 85b]. Compound transactions are a form of atomic transactions that are designed to provide higher concurrency than normal atomic transactions and minimize the problem of cascaded aborts. This implies that the atomic transaction mechanisms provided by the Alpha kernel should permit the relaxation of such constraints as serializability and should permit the use of compensating actions as opposed to returning to previous versions in reaction to aborts. To reduce life-cycle costs, the atomic transactions supported by Alpha should also exhibit a high degree of modularity (i.e., clients should not be greatly inconvenienced when new operations or objects are to be added). Furthermore, since the Alpha kernel's application domain is real-time process control systems, the atomic transaction mechanisms provided by the kernel should bound the amount of time required for transactions to terminate (i.e., either commit or abort).

2.2.3.3. Mechanisms

The common definition of atomic transactions represents a single data point in a multidimensional decision space. The Alpha kernel provides mechanisms that support a range of definitions of atomic transactions by providing some degree of movement along all of the dimensions of this decision space. The atomic transaction mechanisms provided by the Alpha kernel are not completely orthogonal and some points in this space are not meaningful, however these mechanisms represent policy decisions that do not restrict the exploration of the atomic transaction design space. The kernel does not force the client to use a particular form of transactions, but rather allows the client to choose when, where, and what type of atomic transactions to use, based on the level of functionality and the associated cost.

In the context of the Alpha kernel, atomic transactions can be thought of as forming (potentially nested) brackets around portions of threads. Each thread in the system may be executing within a (nested) atomic transaction or outside of any atomic transaction at any point in time. The kernel provides mechanisms that may be used by threads to define when a thread is to enter an atomic transaction and when the thread is to exit the atomic transaction (either by committing or aborting it).

In Alpha, the definition of atomic transactions is decomposed into three separate attributes each of which is supported by one or more mechanisms. These attributes are:

- **Permanence** — which dictates whether the secondary storage image of an object is maintained in volatile storage and does not persist across failures of the node that contains the object's primary memory image, or whether the secondary storage image is kept in non-volatile secondary storage and is regenerated following failure of its node.

- **Failure Atomicity** — which ensures that changes to the secondary storage image of objects are made atomically with respect to system failures. It requires that object updates be done in such a way as to allow the objects to be reconstituted in a consistent state after node failures. This attribute provides the *all-or-nothing* property of atomic transactions by governing the way in which the secondary storage image of an object is modified by atomic updates. This is the functionality typically implemented using stable storage mechanisms [Lampson 81].

- **Serializability** — which provides the appearance (to external observers) that all atomic transactions execute in a non-overlapping serial order. It is not necessary that the activities of threads be actually serialized, only the effects need be equivalent. This attribute involves the control of the visibility, both inside and outside of atomic transactions, of changes made to objects by threads. The attribute of serializability ensures that changes made to an object by a thread are not made visible to other threads until the transaction commits (in which case the changes are made visible to all threads) or aborts (in which case the changes are undone).

The mechanisms in the Alpha kernel that support atomic transactions fall into three categories. Some mechanisms are provided solely for the support of atomic transactions, others are variations on (or extensions to) existing mechanisms, and yet others are general purpose mechanisms that are useful in implementing atomic transactions. Explicitly in support of atomic transactions is a kernel-provided transaction management object that has defined on it operations to begin, commit, and abort atomic transactions. A thread invokes a **BEGIN_TRANSACTION** operation on the transaction management object when it wishes to enter an atomic transaction, either from outside any transaction or from inside another transaction (i.e., in a nested fashion). Threads invoke an **END_TRANSACTION** operation on the transaction management object when they wish to exit the current level of atomic transaction. The **ABORT_TRANSACTION** operation is invoked on the transaction management object when a thread wishes to abort the current atomic transaction that it is executing within. A restriction (that can be

enforced at compile-time) on the use of atomic transactions in Alpha is that the operation invocations to begin, end, or abort a specific transaction must exist within the same operation in an object.

The general purpose mechanisms that are also used to support atomic transactions include the different object types (i.e., transient/permanent, atomic/non-atomic update), the thread synchronization mechanisms, and the invocation mechanism. The permanent and atomically-updated object types are used to provide the attributes of permanence and failure atomicity, and the concurrency control mechanisms are used to provide the serializability attribute. The invocation mechanism is used to perform the COMMIT operation on all of the instances of the transaction manager involved with a particular atomic transaction.

Each object in Alpha has defined on it a set of standard operations for pre-committing, committing, and aborting atomic transactions. A client may provide his own specialized COMMIT and ABORT operations or, if these operations are not specified by the client, the kernel-provided set of default operations is used. By providing custom COMMIT and ABORT operations, the client can define special functions (such as compensating actions) that make use of less expensive mechanisms that are closely tailored to the semantics of the operations being performed in order to maximize performance. In cases where compensation can be done, it is possible to make use of this feature of the Alpha kernel to implement compound (or other non-serializable) transactions [Sha 85a]. The default operation for transaction pre-commit prepares to write the committed state of the object to secondary storage by invoking a PREPARE_UPDATE operation on the object. The default COMMIT operation releases all of the locks and semaphores associated with the transaction being committed, and invokes a COMPLETE UPDATE operation on the object. The default ABORT operation restores all of the locked data items from their logs, releases all of the locks and semaphores associated with the transaction being aborted, and invokes a CANCEL_UPDATE operation on the object.

2.2.3.4. Usage

Typically, atomic transactions have the attribute of *permanence*, in the sense that failures can occur and the changes made to objects by committed atomic transactions remain in effect across failures of system components. This represents an extreme case that provides a high degree of reliability at a commensurately high cost in terms of performance. In Alpha, atomic transactions may have differing degrees of permanence associated with them, thereby providing less than total permanence at less than the worst-case cost. This means that a transaction may commit, but some types of failures will result in the effects of committed atomic transactions being lost.

The attribute of failure atomicity is supported by the object update mechanism and the invocation mechanism. By defining an object to be atomically updateable, the object takes on the attribute of changing states atomically with respect to failures. This function ensures that the secondary storage image of objects is always consistent, and is a necessary part of the failure atomicity attribute. In addition to providing atomically updateable objects, the kernel must be able to ensure that all or none of the actions contained within an atomic transaction commit. This atomicity function is supported by the Alpha kernel's invocation mechanism. When a thread executing in a transaction breaks, in addition to the normal *thread repair* that is done as a part of all invocations, a function known as *visit notification* is also performed by the invocation mechanism on behalf of the thread that has been broken. Visit notification requires that the kernel track all of the objects visited by the thread while in a transaction, and signal these objects (along with the transaction management object) when a transaction commits or aborts.

Synchronization atomicity is attained by Alpha's concurrency control mechanisms. In the absence of some form of concurrency control there are no constraints on the visibility of changes to objects, and hence all changes made to an object are instantaneously visible to any thread executing in that object. The concurrency control mechanisms provided by the Alpha kernel allow threads to control the visibility of changes made to objects. For example, by inhibiting thread access to the sections of an object's code that reads a particular collection of data, a thread may restrict visibility of changes it makes to the data until an atomic transaction commits. In many cases, by taking the semantics of an object's operations into account, a high degree of concurrency can be obtained by threads within objects. Furthermore, the client is free to use the visibility controls in such a way as to maximize use of the concurrency available in the implementation of objects.

The attribute of serializability is provided by enforcing a discipline on concurrent actions, through the careful use of concurrency control mechanisms. The locking mechanism is the primary means by which the serializability attribute is provided in the Alpha kernel. Through a two-phase discipline on the use of locks, serializability of atomic transactions may be achieved. Two-phase locking is only one means by which the goal of serializability may be achieved. Once again, by considering the operations performed by an object, it is possible to achieve the desired effects of serializability at a much lower cost in terms of performance.

2.2.3.5. Issues

Operations may be invoked on objects concurrently by threads, both within and outside of transactions. This results in the possibility of threads within transactions being able to operate on data that has not been committed (i.e., transferred to the object's secondary storage image). In order to provide the attribute of serializability, despite changes made to objects by threads (both within and outside of transactions), all data items locked by an atomic transaction must be written to the object's secondary storage image. This is true regardless of whether the data is modified by this transaction or not (i.e., even data that is locked in read-mode must be committed).

A major concern with the use of atomic transactions is the restrictions that they place on the degree of concurrency that can be obtained from object implementations. Because distributed computer systems provide the opportunity to exploit the concurrency available in applications, any restrictions on concurrency are undesirable. One of the major goals of the Alpha kernel is to permit the exploration of highly concurrent forms of atomic transactions. It has been shown that to increase the concurrency available with atomic transactions, semantic information about the actions being performed must be provided [Garcia-Molina 83]. This semantic information is difficult to automatically derive from programs (however the object model helps out somewhat in this regard).

Some work has been done in the area of loosening the constraints of serializability. In particular, a form of atomic transaction has been proposed, that offers a high degree of modularity and concurrency and promises to be practical in providing failure management and recovery within a real-time operating system [Sha 85a]. The atomic transaction mechanisms in the Alpha kernel provide the means for the validation of these claims. The decomposition of atomic transaction mechanisms in Alpha allows the relaxation of some atomic transaction constraints. For example, the relaxation of the constraint of serializability can be accomplished by unlocking data items prior to committing the transactions in which they were locked. Additionally, to further exploit the potential concurrency in an application, compensating actions can be used in place of the traditional *roll-back* type of abort operations in objects.

In order to place upper bounds on the amount of time it takes for atomic transactions to commit or abort, the invocation mechanism in Alpha includes a means of autonomously detecting node failures and eliminating orphans. This is done by having each node keep track of the state of the nodes up- and down-stream of each invocation it is a part of, and if one of these nodes is found to have failed, the operations on the node involved in that invocation chain are terminated. This bounding of commit time is accomplished at the cost of increased communication overhead [McKendry 85].

2.2.4. Object Replication

In order to meet its goal of availability of services, the Alpha kernel provides support for the replication of objects. The support provided for object replication does not, in itself, constitute a complete data replication scheme. The kernel provides a framework for experimentation in the area of replicated object management. The approach to object replication taken in Alpha is one based on the use of multiple instances of a particular type of object, all of which share a common logical identifier, and consequently appear as replicas of the same object. The general issue of object placement is considered a higher-level issue in Alpha, and therefore the question of how the replicas are to be distributed among physical nodes is to be addressed at a level above the kernel.

As with atomic transactions, the area of object replication is currently being explored as a part of an ongoing thesis project [Shipman 87]. The mechanisms currently provided by the kernel are meant to be representative of a much more comprehensive set of mechanisms to be developed as a result of this work.

The Alpha kernel is designed to be a framework to support a wide range of replication mechanisms, among which are currently supported two forms of object replication — *inclusive* and *exclusive*. In the inclusive form of replication the replicas of an object function together as a single object. Therefore, an operation invoked on an inclusively replicated object (in general) results in the operation being performed on all of the existing replicas. The intent of this mechanism is to increase the availability of an object through a redundancy technique similar to an *available copies* replication scheme [Goodman 83].

To reduce the cost associated with such a replicated invocation, a *quorum* method may be used [Herlihy 86]. Currently in Alpha, a simplified quorum technique is used, where a quorum is defined to be the minimum number of replicas that must involved in an operation before an invocation may be completed. By associating a quorum with each operation defined on an object, fewer than the currently existing set of replicas can be involved in a particular operation. This mechanism can be used by the client to create objects with different degrees of availability, response time, and consistency.

A number of issues related to inclusive replication have been deferred in an effort to reduce the scope of this effort. For example, the timestamp or version number mechanisms needed to implement a proper quorum-based replication scheme have not been provided. Nor has the issue of the regeneration of failed replicas been addressed. These issues are being dealt with in a related thesis project [Shipman 87].

For exclusive replication, the kernel also provides a number of replicated instances of an object type. In this case, however, an invoked operation need only be performed on any one of the replicas for the operation to be considered complete. This approach provides a form of replication in which any one of a pool of undifferentiated object replicas is chosen, based on a global policy designed to meet certain goals. Examples of replica selection policies are "select the first replica that responds", "select the replica that exists at the least loaded node", "select a replica near the invoking object", etc.

The initial policy used to select a replica chooses the first replica to respond to an invocation. This policy provides the potential for a higher degree of both availability and performance than is achievable with non-replicated objects. However, this is accomplished at the cost of not maintaining the consistency of data across the individual replicas.

3
Kernel Interface

The Alpha kernel provides an interface that supports the programming abstractions defined in the previous chapter. The kernel supports these abstractions through a collection of kernel-provided objects, and object programming language-provided mechanisms.

Fundamentally, the kernel interface consists of the object invocation mechanism and a collection of kernel-defined objects. The operation invocation mechanism is made available by the kernel as the only *system call* (implemented with a trap instruction), while the kernel-provided objects are *wired* into the kernel (and are available as soon as the kernel starts up on a node).

Furthermore, each time an instance of the kernel is started up at a node, a permanent object (known as the ***Initialization*** object) is consulted to determine which objects and threads should be instantiated at the given node. The invocation mechanism, the kernel-provided objects, and the ***Initialization*** object and thread, together, serve to break the circularity involved in bootstrapping the system. The ***Initialization*** object is used by the initialization thread to create the desired initial objects and threads for a node at restart, the object and thread management objects provide the means of dynamically creating additional objects and threads, and the invocation mechanism provides the means by which the services provided by these management objects can be obtained by other objects.

3.1. Support for Basic Abstractions

The bulk of the kernel's interface is involved in providing support for the basic abstractions of objects, operation invocation, and threads. The operation invocation mechanism is provided to the client as a programming language construct, and the fundamental support for objects and threads is provided by kernel-provided management objects. In order for the object and thread management objects to be accessed by clients, they may be declared as well-known objects.

Another portion of the Alpha kernel's interface is provided by a set of standard (i.e., kernel-defined) operations on objects and threads. These operations are provided by the kernel for each object, and are used to manipulate the representations of objects and threads. In many cases, the client can specify custom replacements for these standard operations. This permits the client to enhance the characteristics of an object (e.g., its concurrency, performance, etc.) by providing special operations that take advantage of knowledge of the semantics of an object's operations.

3.1.1. Object Management

In support of objects, the kernel provides an object for the dynamic management of objects, and provides a set of standard operations defined on all objects.

3.1.1.1. Object Manager Object

The object programming paradigm used in Alpha suggests that all manipulations of an object be performed by the invocation of operations on the given object. Because it is not possible to invoke an operation on an object that does not yet exist, this approach cannot be used for object creation. For this reason, the kernel provides the ***ObjectManager*** object to permit instances of objects to be dynamically created and deleted.

The ***ObjectManager*** object has two operations defined on it for creating new objects — one to create simple instances of object types, and an other to create multiple instances of an object, all with the same logical name. The latter operation represents the kernel-level support of the object replication objectives of the kernel. Because the object placement function belongs above the kernel, an additional, higher-level facility is required for the complete management replicated objects.

The ***ObjectManager*** object has the following operations defined on it:

CREATE: This operation is used to create new instance of objects. The parameters of this operation specify the type of object that is to be created, and the optional attributes that the new object is to take on. New instances of the specified type of object are created at the node where the invocation is made. This operation returns a capability for the newly created object, along with the rights associated with the new object. This operation will fail if the specified object type is unknown, if the specified attributes are invalid, or if the node currently lacks the resources necessary to support another object.

REPLICATED_CREATE: Like the previously defined operation, this operation is used to create instances of objects. This operation, however, takes an additional set of parameters and creates multiple instances of an object of the specified type. The additional parameters

are used to specify the number of replicas that are to be created, the type of replication to be used with the new replicated object (i.e., inclusive or exclusive), and an optional indication of the quorum size for each operation defined on the replicated object being created. This operation returns a single capability for the newly created object, and the specified number of objects replicas are in fact created at the local node. It is up to the higher level replication management functionality to control the placement of the object replicas.

DELETE: This operation is used to delete an existing object instance. The operation takes as parameters a capability for the object to be deleted and an indication of whether the object should be deleted even if threads are currently active within the object. If the capability is valid and the thread activity condition is met, this operation deletes the specified object and deallocates the resources associated with it. The specified object is deleted regardless of its physical location in the system.

3.1.1.2. Object Standard Operations

The kernel defines a set of standard operations that the client can invoke on objects. Should the client provide an operation with the name of a standard operation, it will be used in place of the kernel-defined operation.

The kernel-defined, standard operations on objects are:

STOP: This operation halts all of the threads currently executing within the object. Once in the halted state, the kernel does not allow any further invocations to be performed on an object. This is useful in debugging and in preparing an object for migration among nodes.

START: This operation allows restores the specified (halted) object to its previous state (i.e., allowing normal execution to continue).

UPDATE: This operation is used to write (portions or all of) the primary memory images of objects to their secondary storage images. This operation takes a parameter that indicates the type of update operation to be done. The different types of update are: *prepare* update, that returns once the object's image has been written to secondary storage in such a way that the update operation can be completed even if node crashes should occur; *complete* update, that returns only when the object's image has been completely written out to the secondary storage image — it may be used following a *prepare* update operation to actually update the object's secondary storage image, or it can be used to force a complete update on an unprepared object; *cancel* update, that is used following a *prepare* update operation to cancel the attempted update. This operation behaves differently depending on whether the object has the atomic update attribute associated with it.

MIGRATE: This operation is used by the client to move objects among the nodes in the system. This operation takes a parameter that identifies the node to which the object should be moved. There are restrictions applied to the time at which

objects can be migrated (i.e., the object must be in a halted state), and the operation returns only when the requested movement has been completed.

MODIFY: This operation is used to modify the attributes of an existing object. This operation takes as a parameter the new attributes that the object should assume. The object attribute parameter is a structure consisting of a list of name/value pairs, with one for each attribute that is to be modified.

PRE-COMMIT: This operation is used in the first phase of a two-phase commit function. It performs a *prepare*-type update operation on the object, and returns a success indication.

COMMIT: This operation is used in the second phase of a two-phase commit function. It performs a *complete*-type update operation on the object, (in the case of a top-level commit) releases any semaphores or locks held by the object, and returns a success indication.

ABORT: This operation is used to abort a transaction, either before or after the pre-commit phase. This operation restores all locked data items to their previous state, performs a *cancel*-type update operation on the object, releases any semaphores or locks held by the object, and returns a success indication.

3.1.2. Thread Management

In support of threads, the kernel provides an object for the dynamic management of threads, and provides a set of standard operations defined on all threads.

3.1.2.1. Thread Manager Object

The kernel provides an object, known as the ***ThreadManager*** object, that permits the dynamic creation and deletion of threads. The ***ThreadManager*** object provides functionality analogous to that of the previously defined ***ObjectManager*** object.

The ***ThreadManager*** object has the following operations defined on it:

CREATE: This operation is used to create threads. The parameters required of this operation are a capability for the object, the name of the operation that the new thread should begin execution within, the initial characteristics of the thread, and any parameters that might be called for by the specified initial operation. This operation returns a capability for the newly created thread, along with all of the rights associated with threads. If the specified object does not currently exist on the node at which this operation is invoked, the operation fails, and an indication to this effect is returned to the invoking object.

DELETE: This operation is used to delete threads. A capability for a thread is given as a parameter. Provided that the given capability is valid, this operation deletes the

specified thread and deallocates the resources that are currently associated with it. Furthermore, the thread specified by the given capability is deleted regardless of its current location in the system.

3.1.2.2. Thread Standard Operations

The kernel defines a collection of standard operations defined on threads. These standard operations are similar to those defined by the kernel on objects, however it is not possible for the client to specify custom versions of these operations.

The kernel defines the following operations on all threads:

STOP: This operation suspends the logical progress of the thread.

START: This operation causes the execution of a thread that was halted by a **STOP** operation to be be resumed.

MIGRATE: This operation is used by the client to move threads among the nodes in the system. This operation takes a parameter that identifies the node to which the thread should be moved. There are restrictions applied to the time at which threads can be migrated (i.e., the thread must be in a halted state), and the operation returns only when the requested movement has been completed.

SLEEP: This operation causes the progress of a thread to be suspended for a specified period of time. This operation requires a parameter that indicates the amount of time the thread is to sleep for. The time parameter is an integer which represents the amount of time to delay the thread in units of microseconds.

MODIFY: This operation is used to modify the attributes of the thread. It requires a parameter that describes the new attributes that the thread should take on. The thread attribute parameter is a structure consisting of a list of name/value pairs, with one for each attribute that is to be modified.

DEADLINE: This operation modifies the thread's environment to indicate the time critical nature of the computation that is to be performed by the thread. Each invocation of a **DEADLINE** operation must have a matching invocation of a **MARK** operation within the same object and operation. This operation takes parameters that indicate the time by which the matching **MARK** operation must be executed, the estimated amount of time that should be required to reach the matching **MARK** operation, and whether continuing the computation has any value to the system if the deadline is missed.

MARK: This operation indicates the end of a time-critical computation for a thread. The **DEADLINE/MARK** operation pairs may be nested, within the same operation, or within subsequently invoked operations. The **DEADLINE** and **MARK** operations are provided to the client as a single, block-structured construct by the object programming language pre-processor.

3.2. Support for Additional Abstractions

The kernel provides other objects in support of the Alpha kernel's abstractions. Among these are objects that provide thread concurrency control, an object for the management of atomic transactions, and an object for the initialization of individual instances of the kernel.

3.2.1. Semaphore Manager Object

The kernel provides the ***SemaphoreManager*** object in order to permit other objects to create and delete individual instances of ***Semaphore*** objects. ***Semaphore*** objects are bound to their creating object, each newly created ***Semaphore*** object can only be used by the object that created it, and likewise the DELETE operation can only be invoked on a ***Semaphore*** object by its creator.

The operations defined on the ***SemaphoreManager*** object are:

CREATE: This operation instantiates an object of the kernel-defined type ***Semaphore*** and associates it with the invoking object. This operation requires a parameter to indicate the initial value that the semaphore's counter should take on. This operation returns a capability for the newly created ***Semaphore*** object to the invoking object, with the NO_TRANSFER and NO_COPY restrictions applied to its operations.

DELETE: This operation deletes the ***Semaphore*** object whose capability is passed as a parameter to this operation. Because the NO_TRANSFER and NO_COPY restrictions are associated with the capabilities given to the ***Semaphore*** object's creator, only the object that created a semaphore has the right to perform the delete operation on it. A second parameter is required by this operation, which is an indication of whether an error should be returned if there are threads blocked on the ***Semaphore*** object to be deleted, or if any blocked threads should be unblocked and the object deleted in any event.

3.2.2. Semaphore Object

Semaphore objects are instances of a kernel-defined object type, that are created and deleted by the ***SemaphoreManager*** object. Instances of ***Semaphore*** objects are used to control the concurrent execution of threads within objects. A ***Semaphore*** object provides functionality equivalent to the counting semaphore constructs found in many other systems. Concurrency control is achieved by blocking and unblocking threads with these semaphores. In addition to the basic P and V operations, a conditional P operation is provided to allow a non-blocking type of synchronization.

The operations defined on ***Semaphore*** objects are:

P: This operation performs a standard P operation on a counting semaphore. Logi-

cally, this operation can be considered to represent an attempt at obtaining a token that corresponds to a resource that the ***Semaphore*** object manages. If a token is not available, the thread that invokes this operation is blocked until one becomes available. Once a token has been granted and the resource used, the thread must then invoke a V operation (to, logically, return the token).

C_P: This operation performs a conditional P operation on a counting semaphore. This operation is similar to the previously defined **P** operation, except that if a token is not available, the thread is not blocked, the semaphore's count value is not decremented, and a failure indication is returned. If a token is available, however, this operation grants it to the invoking thread, and the semaphore's count is decremented. If a **C_P** operation returns a success indication, it must also invoke a **V** operation (to return the token) when execution in the critical section is complete.

V: This operation performs the standard V operation on a counting semaphore. Logically, this operation represents the return of a previously granted token to the semaphore's resource pool. If there are any threads blocked waiting on a token, the thread that has been blocked the longest is granted the token and unblocked.

3.2.3. Lock Manager Object

The kernel provides the ***LockManager*** object in order to support the dynamic creation and deletion of individual ***Lock*** objects. As with ***Semaphore*** objects, ***Lock*** objects are bound to their creating object. Each ***Lock*** object, therefore, can only be used by the object that created it, and likewise the **DELETE** operation on the ***LockManager*** object can only be performed on a ***Lock*** object by the object that created it.

The ***LockManager*** object has the following operations defined on it:

CREATE: This operation creates a new instance of a ***Lock*** object and associates it with the invoking object. This operation takes as parameters a pointer to the start of the data item within the invoking object, and the size (in units of bytes) of the data item to be locked. This operation returns a capability for the new object (with NO TRANSFER and NO_COPY restrictions), and a failure indication is returned if the specified data item is invalid (i.e., conflicts with an existing lock, not in the objects data region, insufficient kernel resources, etc.).

DELETE: This operation deletes an instance of a ***Lock*** object. The operation takes as a parameter the capability of the ***Lock*** object to be deleted. Furthermore, a second parameter is required to indicate if an error should be returned if there are threads blocked on the ***Lock*** object to be deleted, or if any blocked threads should be unblocked and the object deleted.

3.2.4. Lock Object

Lock objects are instances of a kernel-defined object type, and are created and deleted by the ***LockManager*** object. The purpose of ***Lock*** objects is to control the concurrent access of threads to individual data items within objects. When a ***Lock*** object is created, the data region with which the lock is associated is specified, as well as the mode in which the lock is to be acquired in (which reflects the type of operation to be performed on the data associated with the lock). A *lock compatibility table* specifies which modes of access are compatible and which constitute conflicts.

As with the ***Semaphore*** objects, concurrency control is achieved by blocking and unblocking the threads that attempt to lock data items. A conditional lock operation provides non-blocking functionality similar to the C_P operation defined on ***Semaphore*** objects.

It is possible to change the mode in which a lock is being held by invoking another lock operation prior to performing the unlock invocation. This has the effect of atomically invoking an unlock operation followed by a lock operation with the new lock mode. Furthermore, an operation is provided on ***Lock*** objects to alter the scope of a ***Lock*** object without having to delete and recreate the lock.

Lock objects also figure prominently in the support of atomic transactions in Alpha. However, the features of ***Lock*** objects that contribute to the functionality of atomic transactions are transparent to the users of these objects.

Lock objects have the following operations defined on them:

LOCK: This operation attempts to lock the data item related to the lock, in a particular mode. The mode that is passed as a parameter to this operation represents the logical operation that the invoking thread wishes to perform on the data item to which the ***Lock*** object corresponds. The lock compatibility table is consulted, and the thread is blocked until that point at which no conflicting locks are held.

C_LOCK: This operation is similar to the normal LOCK operation, but does not ever result in the invoking thread being blocked. Instead, a success or failure indication is returned, based on the current state of the lock (i.e., what modes have locks been granted in) and the lock compatibility table. The state of the lock is only changed when a compatible operation is invoked.

UNLOCK: This operation releases the lock held by the invoking thread. The result of this operation is to return the lock to its previous state, determine if there are any threads blocked that are waiting to make requests that are now compatible, grant the lock to the thread with a compatible lock request that has been waiting the longest, and unblock the chosen thread.

MODIFY: This operation is used to alter the scope of the data item with which the ***Lock*** is associated. A parameter is given that is interpreted as the new size of the data item with which the ***Lock*** object is associated. While the size of the data item can be changed by this operation, its origin must remain the same.

3.2.5. Transaction Manager Object

The ***TransactionManager*** object is provided by the kernel in support of atomic transactions. This object is used by objects to indicate to the kernel when a transaction is to begin, to end, or to be aborted. The transaction manager carries out the site coordination function and permits the client to specify the particular type of transaction to be used.

In the current implementation of the Alpha kernel only nested, traditional atomic transactions are supported, where the begin, end, and abort operations for individual atomic transactions appear within the same operation. Provisions have been made, however, to support such types of transactions as compound transactions.

The following operations are defined on the ***TransactionManager*** object:

BEGIN: This operation indicates to the kernel that the invoking thread should enter a new level of atomic transactions. This operation takes a parameter indicating the type of atomic transaction that should be initiated. The thread invoking this operation acquires the attribute of being in a transaction, at the current depth.

END: This operation is the client object's indication to the kernel that the current level of atomic transaction should be concluded. While the behavior of this operation differs based on the type of atomic transaction, the commit operation for a *top-level* transaction (i.e., one that is not nested within another transaction) involves the following: **UPDATE** operations are performed on the objects modified by the transaction, and the locks acquired in the course of the transaction are actually released. All of this behavior is extended to objects that are nested within the transaction being committed as well. Furthermore, if the transaction being committed by this operation is not a top-level transaction, the updates are not made, nor are locks released.

ABORT: This operation is used to indicate to the kernel that the current level of atomic transaction is to be aborted. The result of this operation is dependent on the type of the atomic transaction, however in the case of basic transactions, the following is performed: any ***Semaphore*** objects on which there are outstanding **V** operations, have **P** operations performed on them, all locks acquired by the transaction are released and the locked data items that were modified during the course of the transaction are restored to their original state, and the thread's execution is diverted to the statement following the transaction's **END** operation.

3.2.6. Initialization Object

The kernel provides the ***Initialization*** object to deal with node restart issues. While this object is not in direct support of any of the Alpha programming abstractions, it is necessary to provide an orderly start-up when a node comes on-line — either initially on power-up or following a failure restart.

The ***Initialization*** object is a permanent and atomically-updated object that becomes available as soon as a kernel instance initializes itself. Once a kernel instance has been initialized, all outstanding updates have been made to secondary storage at the node, and all of the permanent objects that exist at the node have been reconstituted, the objects and threads specified by the ***Initialization*** object are created. The ***Initialization*** object first instantiates the objects, followed by the threads, that it was instructed to create.

The operations defined on the ***Initialization*** object are:

ADD: This operation is used to add an object or thread to those that are to be instantiated when the local node restarts. This operation takes as a parameter a capability for the desired object or thread. If the entity to be added is a thread, an additional pair of parameters is required to indicate within which object and operation the thread is to begin execution.

REMOVE: This operation is used to remove an object or thread from the set of those which are to be instantiated when the node at which the invocation has been made restarts. This operation takes as a parameter a capability for the object or thread to be removed.

3.3. Supplementary Programming Constructs

In addition to the kernel-provided objects previously described, the Alpha kernel's interface includes a set of mechanisms provided to the client via a set of programming language constructs. While most of the Alpha kernel's functions are provided by invoking operations on kernel-defined objects, the simple object language pre-processor used in Alpha provides access to a limited set of kernel mechanisms via language constructs. All objects to be executed on the Alpha kernel are written in the C programming language with a small set of extensions to support the style of object-oriented programming used in Alpha. Appendix A contains a description of these language extensions.

4
Kernel Functional Design

This chapter describes the design of the various components of the Alpha kernel and enumerates the major design decisions that shaped the kernel. Throughout, an attempt is made to provide a justification for the decisions.

Because one of the major goals of the Alpha kernel is flexibility, great emphasis has been placed on the design of mechanisms — as opposed to the specific policies concerning their use. There are many instances throughout the kernel where simple policy decisions have been made, not because they represented the best design decisions, but because it was necessary to make some sort of decision in order to permit progress to be made on the development of the rest of the system. In some cases there are definite plans for replacing these simple policies with better ones, but in other cases the decisions reflect a lack of emphasis dictated by the research interests of the overall research project.

A number of functions necessary in the creation of reliable, distributed, real-time systems are not provided by the kernel. In these cases, the functions are supported by kernel mechanisms, but the specific policies are applied at higher levels in the system. An example of this is in the area of object location management. Ultimately, the function of dynamic reconfiguration is to be performed by system-level facilities that manage the physical location of objects. At this time, however, the kernel supports this functionality by permitting objects to be created on the node at which the creation request was made and providing a mechanism that allows objects to be moved among nodes.

This chapter begins with a discussion of the design of the mechanisms that support each of the programming abstractions provided by the kernel. Next is a description of the design of the major facilities that support the kernel's functionality. Finally, there is a description of the various optimizations that were included in the design of some of the mechanisms in the Alpha kernel.

4.1. Basic Mechanisms

This section outlines the design of each of the major mechanisms that support the client's programming abstractions provided by Alpha. The Alpha kernel does carry the client-level abstractions all the way through the implementation of the system (as do some systems [Levy 84, Cox 83]), but rather it uses more conventional techniques within the kernel to provide the system abstractions. This is not to say, however, that the abstractions supported at the client interface are not also available for use within much of the kernel. Mechanisms such as operation invocation are used throughout the kernel's internals.

4.1.1. Objects

The design of objects in Alpha closely mirrors the object abstraction presented to the client. Objects consist of regions of data that the object serves to encapsulate, regions of code that are used to perform the operations defined on the object, and the various control structures used by the kernel to manage the object. Figure 4-1 illustrates the layout of an object within a virtual address space in Alpha.

4.1.1.1. Design

In this design, the data portion of objects consists of three sub-parts: statically allocated, uninitialized data; statically allocated, initialized data; and dynamically allocated, uninitialized data (i.e., heap storage). The data part of an object represents only the global data associated with the object; all local data (e.g., automatic variables of subroutines) are provided on a per-thread basis and are not associated with the object proper.

Objects in Alpha are designed to be quite similar to the code and data portions of traditional processes (i.e., a process without a stack segment or process control block). Additionally, each object resides in a separate virtual address space (or *context*), allowing the abstraction of disjoint object memory addressing domains to be enforced by a processing node's memory management hardware. Addresses generated within an object cannot reference memory locations in any other object's memory address domain. Furthermore, each portion of an object is also similarly protected by the memory management hardware; the code portion is protected *execute-only* and the data portion is *read/write* protected. To ensure that execution within an object's code portion can begin only at the start of an operation's code, the specified operation entry points within an object's code portion are also designed to be hardware protected. While most operations are client-specified and appear in the code portion of the object, the code for standard operations is part of the kernel and is shared by all objects.

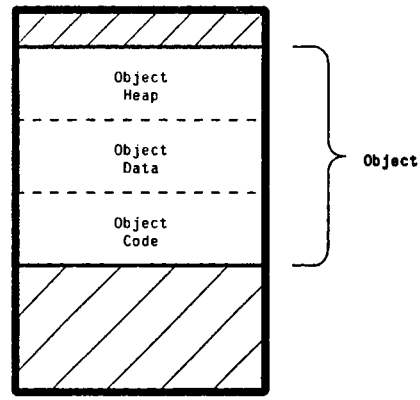

Figure 4-1: Object Structure

At any point in time the executable image of an object resides entirely on a single node, although at times portions of an object's image may be in secondary storage because of paging or swapping activities. When a thread executing within an object invokes an operation on another object, the target object is mapped into the address space of the invoking thread. When an operation on an object is complete, the invoked object is replaced with the object that made the invocation. The concurrent access of objects by threads makes it possible to have an object's image be shared by more than one context at a time (all at a single node).

To instantiate an object, an invocation is made on the ***ObjectManager*** object, specifying the type of object to be created. Object types are templates, maintained in secondary storage, that contain the code and initialized data for objects and are used in instantiating new objects. The act of instantiating an object involves: the construction of the necessary kernel data structures for the new object, the initialization of its virtual memory structures, the allocation of space in the secondary storage subsystem for the object's image, the generation of a unique identifier for the new object, the registration of the new object with the kernel, and the return to the creating object of a capability for the new object. This is all performed on the node at which the instantiation operation was invoked. Once the object has been instantiated, its code and data are loaded on demand, as operations are invoked on the object.

Many traditional system assumptions were made for Alpha, because one of the objectives of Alpha was that it should be implemented on generic, currently available hardware. For example, it was

assumed that objects would exist in and be executed out of the primary memory at each of the system's nodes. A further assumption is that the primary memory is volatile and is not capable of containing all parts of an entire application at once. For this reason, another layer was introduced into the storage hierarchy to provide both the permanence lacked by the primary memory, and to extend primary memory into higher capacity (and lower performance) storage.

In Alpha, an image of each object's primary storage representation is maintained in secondary storage to provide objects with a number of desired characteristics. The traditional extension of physical memory is accomplished by paging objects between their primary and secondary memory images — that is to say, object images are divided into pages and are brought into primary memory when they are accessed (i.e., on demand); and, when necessary, pages are moved out to secondary memory according to a simple FIFO-like paging algorithm.

The image of an object that is maintained in secondary storage may not exactly represent the state of the object at all times, instead it serves as a repository for a representation of the object whose exact meaning can be defined by the client. Furthermore, several of the optional attributes of objects are defined in large part by the manner in which the secondary storage image of objects is managed. For example, the secondary storage images of objects with the transient attribute are stored in the part of secondary storage that is volatile, while the images of objects with the attribute of permanence are maintained in the non-volatile portion of secondary storage.

Among the standard operations defined on objects, the **UPDATE** operation is the most significant with respect to the object's relationship to its secondary storage image. The **UPDATE** operation carries out a checkpoint-like function that is used to make an object's secondary storage image reflect the current state of the object. The **UPDATE** is performed using a version of the object's primary memory state in which all of the object's data conforms to the consistency constraints associated with the object (i.e., the committed state of the object). If there are *write-locks* held on the object, the data items associated with these locks could be in an inconsistent state, so the update is performed using the guaranteed consistent (i.e., committed) state of the data kept in the log areas associated with the locks.

The **UPDATE** operation has a similar effect on objects with either the transient or permanent attributes; the only difference between the two is that transient objects are not reconstituted following a failure and therefore having a consistent secondary storage image is of little value. On the other hand, the **UPDATE** operation has a very different effect if an object has the attribute of being atomically updateable. With non-atomically updated objects, the update is performed on the secondary

storage directly, whereas with atomically updated objects, updates are first done to a buffer area in secondary storage and an intentions list is created. This is a type of stable storage technique (e.g., as found in [Sturgis 80]), and is provided to allow atomic update operations to be completed even in the face of node failures.

The different variations of the UPDATE operation are selected by an invocation parameter. The *complete* variant performs a complete update operation on the object before returning. If a node (on which the object or its secondary storage image exists) fails before this operation terminates, the resulting effects depend on the attributes of the object. In the case of objects that are not atomically updateable, an interrupted complete update operation can result in the secondary storage image being partially updated. On the other hand, the secondary storage images of atomically updated objects will either be in their original states (if the crash occurred before the complete intentions list was written), or completely updated when the crashed node(s) recover (if the crash occurred after the intentions list was written). The *prepare* variation of the UPDATE operation prepares an object for update by writing the changes to be made into a non-volatile buffer, creating an intentions list, and then returning without actually performing the update to the object's secondary storage image. The *cancel* variant deletes a previously created intentions list and removes a prepared update from the non-volatile buffer. This is done to release the buffer space being held by the prepared update, when it is determined that the prepared update should not actually be performed. This operation has no effect if a prepare update operation has not been issued. The prepare update operation is used in the first phase of a two-phase commit operation, and is to be followed by either a complete update or a cancel update operation. The complete update operation determines whether the object has previously invoked a prepare update, and makes use of a prepared update where possible. The prepare update and cancel update operations have no effect when the object they are performed on is not atomically updateable.

4.1.1.2. Rationale

A number of benefits derive from this design of objects, including the fact that only a partial context switch is required when an operation is invoked, the separation of object address spaces and entry points into the object's code can be enforced with hardware, and objects can be implemented efficiently on standard hardware.

This design of objects is such that each operation invocation does not require that a full context swap take place — only the object portion of a virtual address space must be remapped when a (local) invocation is performed. The full context swap (including interaction with the scheduler and

dispatcher) that must be performed in most process/message-oriented systems, need not be performed each time an operation is invoked in Alpha. While this design enhances system performance, the current processor's memory management unit is oriented towards processes, and therefore the partial context switch is more costly than it would be with an object-oriented memory management unit.

In Alpha, fault containment is considered to be quite important. Because a system's reliability goals are typically more difficult to achieve than its performance goals, it is considered a good tradeoff to pay the cost of isolating each object in its own address space. With modern processor architectures, there is little justification for having objects share address spaces and give up the benefits of fault containment in favor of the marginal increases in performance gained by limiting interdomain jumps. (Note that language-based systems may be able to provide some degree of object addressing domain separation at compile-time [Morris 73].)

It is also desirable for the kernel to restrict access to the code portion of objects to valid operation entry points. Despite the run-time performance costs imposed, this function provides an important form of defensive protection against the inadvertent, unconstrained execution of the code implements an object's operations. In the Alpha kernel, this access to entry points within objects is controlled by kernel-provided indirection tables. Given the appropriate memory management hardware however, this function could also be supported with much greater efficiency.

The fact that the structure of an object in Alpha is similar to a typical process (without a stack segment) allows objects to be implemented with reasonable efficiency on standard, process-oriented, hardware. Furthermore, this design allows an object to be shared by multiple threads, i.e., exist in multiple contexts simultaneously. This allows the efficient sharing of objects through the virtual memory mapping structure, and provides the potential for greater system-level concurrency.

4.1.2. Operation Invocation

In the Alpha kernel, the movement of a point of control associated with a computation (i.e., a thread) among objects is accomplished through the invocation of operations on objects. In addition to encapsulating code and data, objects provide the entry points for the invocation of operations. Operations may have one or more parameters associated with them, however, the object model of programming suggests that large amounts of data should not be passed as parameters to invocations.

Because the operation invocation facility is intended to be used for all interactions among objects in

the system, the facility is designed to provide, at the lowest possible level in the system, physical-location transparent access to objects. All system-provided services are made available through the operation invocation facility; the kernel routines are designed to provide object-like interfaces to the clients. Various kernel facilities (most notably the virtual memory facility) also make use of the invocation facility as a part of their normal function. This usage of the invocation facility provides the kernel with full visibility of all movements of threads among objects (an important feature in the support of atomic transactions and for system monitoring).

Since the invocation facility is the only means of object interaction in Alpha, it is important that the costs associated with its use be kept at a minimum. However, kernel-provided implementations of enhanced-functionality operation invocation services seem appropriate because of the increases in performance obtainable through downward functional migration, and the non-recurrence of costs associated with system-provided services. The Alpha kernel provides a range of different invocation services, each designed to meet a different set of needs at a cost commensurate with the level of functionality provided. To the clients of the kernel, however, all of the different types of operation invocation service appear the same. It is the kernel's responsibility to determine the specific type of invocation service that is to be used, on a per-invocation basis. In this way, the client need be presented with only a single, conceptually simple invocation primitive, and the details of what is required to deliver the desired semantics are hidden.

This client-level uniformity does not imply a uniform underlying implementation. There is not a single worst-case mechanism that is used for all invocations and provides the highest level of functionality at the highest cost. The kernel ensures that the client receives only the functionality necessary for a given invocation and consequently the client pays only the necessary cost, based on characteristics of the invoking object, the invoked object, and the current state of the thread making the operation invocation.

4.1.2.1. Basic Invocation Service

The basic operation invocation mechanism provides the semantics of a simple Remote Procedure Call (RPC) service, and all of the specialized types of invocation are constructed on top of this basic facility. The simplest form of invocation involves a client-defined object invoking an operation on another client object that exists on the same node.

To perform an operation invocation, the parameters are placed into a special data structure, and the thread traps into the kernel, passing the parameter data structure to the kernel. The parameters of an

invocation consist of a specification of the object and the operation to be invoked, along with the arguments required by the specified operation. Because the invocation facility is the only direct means by which objects can interact with one another, it is the point at which object access restrictions are enforced. The invocation of operations on objects is controlled by the use of capabilities as destination object specifiers.

In Alpha, capabilities define an object, the operations that can be performed on the object, and the manner in which the capability can be used. When an operation invocation is made, the kernel must first check the validity of the target object's capability. This involves determining whether the invoking object owns the specified capability, whether the capability has the rights necessary to perform the desired operation, and whether the desired use of the capability does not violate the restrictions associated with it. Once validated, the capability is translated into an internal global identifier for the destination object that is used as an address for the operation invocation.

The global object identifier is used within the kernel to locate the control structures for the destination object. This is done by performing a lookup operation on the local node's table of currently active local objects (known as the *Dictionary*). If the target object is found in the Dictionary, the kernel manipulates various data structures and maps the invoking object out of the thread's address space, and then maps, in its place, the invoked object. The specification of the operation to be invoked is used to locate the desired entry point into the code of the invoked object. The mapping of an operation identifier to an object entry point is performed in a manner similar to the validation and translation of capabilities.

Once the invoked operation is complete, the invocation procedure described here is reversed, parameters are returned, and the thread continues executing following the invocation statement in the invoking object.

Figure 4-2 illustrates the behavior of the basic operation invocation activity. The figure represents the virtual address space of a thread (i.e., $\mathbf{Thread_i}$), executing within an object (i.e., $\mathbf{Object_a}$) and invoking an operation on another object (i.e., $\mathbf{Object_b}$). In part **(a)** of the figure, $\mathbf{Thread_i}$ is making an invocation on $\mathbf{Object_b}$, from within $\mathbf{Object_a}$. Part **(b)** of the figure represents the virtual address space of $\mathbf{Thread_i}$ from the point at which the invocation on $\mathbf{Object_b}$ is initiated, until the operation completes and a return from the invocation is begun. Part **(c)** illustrates $\mathbf{Thread_i}$'s virtual address space after the completion of the operation invocation.

The parameters passed in operation invocations, either to or from an invoked object, can be either

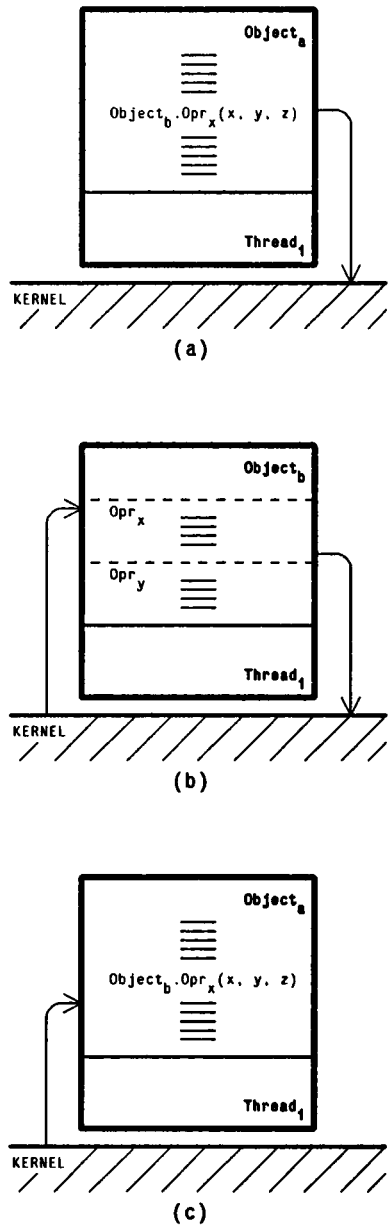

Figure 4-2: Basic Operation Invocation

variables or *capabilities*. All variable-type invocation parameters are passed by value, and any number (within a kernel-defined limit) of this type of parameter may appear in an operation's invocation parameter list. These are similar to the parameters passed by value in standard C language procedure calls, and structured data can be passed as parameters. The Alpha kernel does not perform run-time type checking of variable parameters because it is assumed that the compile-time type checking mechanisms will meet the needs of the system. Also, each invocation returns a status variable that is used to indicate that the invocation succeeded or that the invocation failed, in which case the presumed reason for the failure of the invocation is also returned.

In addition to variable-type parameters, capabilities can also be passed as invocation parameters. Just as with the capabilities used to specify the object that is the target of an invocation, capabilities passed as parameters are first validated to ensure that the invoking object owns the capabilities and is not restricted from passing them in an invocation. Once validated, capabilities passed as parameters are translated into a global format, added to the invoked object's list of capabilities, and then translated into representations of capabilities local to the destination object. A failure indication is returned to the invoking object if any of the capabilities in the invocation are found to be invalid.

The invocation parameter passing mechanism takes advantage of the ability of the system's memory management hardware to expedite the movement of physical memory pages within and among virtual address spaces. The ability to remap physical pages at low-cost makes it possible to avoid the high overhead traditionally associated with the copying of parameter blocks among separate address spaces. In addition to mapping invoked objects in and out of the address space of the invoking thread on invocation, the kernel must move parameter blocks between the invoking and invoked objects. A logical stack of invocation parameter pages is maintained by the operation invocation facility, with the active invocation's parameters always on the top of the stack. This invocation stack is used to keep the parameter block for the currently active invocation mapped into a fixed location in each address space, throughout any series of (possibly nested) invocations.

The parameters passed into, and returned from, an operation invocation are placed in a data structure corresponding to a physical memory page, known as an *invocation parameter page*. Invocation parameter pages are mapped into different locations within the address spaces of threads in order to effect the passing of parameters among objects. Each thread has associated with it a pair of active invocation parameter pages — an *incoming* invocation parameter page that is shared by the current object and the object that invoked it, and an *outgoing* invocation parameter page that is shared by the current object and any of the objects it invokes.

Each invocation parameter page is composed of two parts — the *request* part, that contains any of the parameters to be passed into an invoked object, and the *reply* part, that contains any parameters that are to be passed back to the invoking object upon the completion of the operation. Invocation parameter pages are managed in an *overlapped window* fashion — i.e., when an operation is invoked, the incoming parameter page is mapped out of the thread's context, the outgoing parameter page is mapped into the incoming parameter page location for the invoked object, and a new page is allocated and placed in the invoked object's outgoing parameter page. Figure 4-3 illustrates the behavior of the parameter pages when an operation is invoked. When an invocation completes, this process is reversed. The outgoing parameter page is deallocated, the incoming parameter page is mapped back into the outgoing parameter page's location, and the old incoming parameter page is restored. Should any returned capabilities be found to be invalid, none is returned in the parameter block and a failure indication is returned to the invoking object indicating the cause.

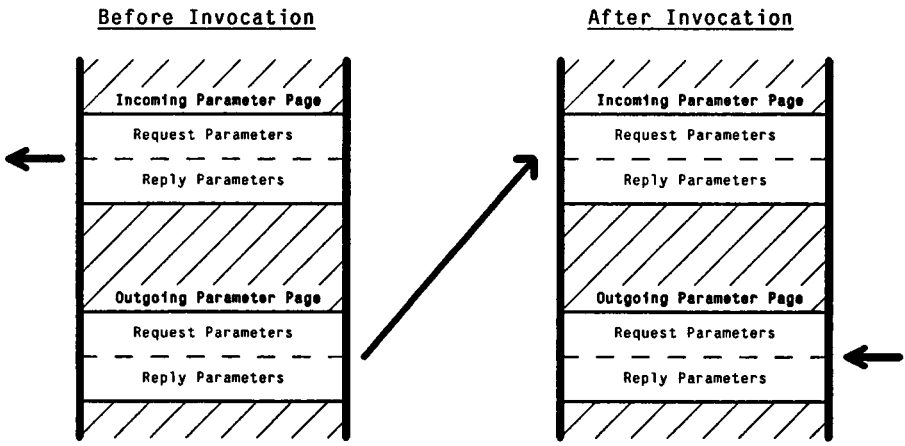

Figure 4-3: Parameter Passing on Invocation

4.1.2.2. Specialized Invocation Services

In addition to the basic service provided by the operation invocation facility, there are a number of specialized services provided by the kernel. The different types of invocation service provided by the Alpha kernel can be categorized by a decision tree consisting of the following (independent) questions:

- Is the target of the invocation a client-defined object, or is it a kernel service provided in the form of an object (i.e., a system call)?

- Is the object that is the target of the invocation physically co-located with the invoking object (in which case certain optimizations that rely on shared memory can be performed), or does the target object reside on another node (in which case the services of the communications subsystem must be employed and information must be moved between nodes)?

- Is the thread making the invocation outside of any atomic transaction (in which case the standard operation invocation semantics are used), or is the invoking thread within an atomic transaction (in which case the invocation semantics must take on the attribute of visit notification)?

- Is the target of the invocation a simple individual object, or is it a replicated object? Furthermore, if a target object is replicated, what type of replication is being used (e.g., inclusive or exclusive)?

— System Object Invocation

This variety of invocation service is used when the target of an invocation is not another client-defined object, but a kernel-provided service whose interface is like that of an object (these types of objects are known as *system service objects*). This kind of invocation service is an efficient means of providing client objects with system services, and it maintains the uniformity of the object model in Alpha (i.e., system services are provided without introducing a new abstraction for system calls).

When the target of an invocation is a system service object, this specialized invocation protocol is used to reduce the overhead associated with performing a normal, client object-to-client object invocation. While system service objects appear as regular objects to clients, the invocation facility detects references to them and short-circuits the usual invocation process. Invocations to system service objects are intercepted by the invocation facility, interpreted, and the appropriate kernel routine is called (with the passed parameters). With this invocation service, the kernel's Dictionary need not be searched to find the target object. The capability corresponding to the destination object is still validated to determine whether the invoking object has the proper rights to make the desired system call, and any capabilities passed in parameters are also similarly validated.

As in the case of basic invocation, the invocation of a system object involves the marshaling of the invoking client object's parameters and then trapping into the kernel. In this case the kernel does not map the invoked object into the client's address space and return the point of control to the invoking thread's context. Instead, operations on system service objects are performed entirely within the kernel, and once the operation on a system service object completes, the thread resumes its execution in the invoking object in the usual fashion.

— Remote Invocation

The basic invocation facility defined above describes only the case in which the target object is local to the invoking object. The remote operation invocation service is called for when the invoked object is on another processing node.

While it would be possible to extend the basic invocation service across multiple nodes (i.e., one could design the kernel in such a way as to have the communication subsystem extend the virtual address space of a thread across multiple nodes), the current design of Alpha takes a more conventional approach. When an operation is invoked and the destination object is not found in the (local) Dictionary, it is assumed that the object is on another node and the communication subsystem is used to locate the desired object. If the communication subsystem does not return an indication that the target object was located in some period of time and after some number of retry attempts, it is assumed that the invoked object does not exist and an indication to this effect is returned to the invoking object.

When an invoked object is successfully located on a remote node, a context is allocated on the remote node and initialized to serve as a remote version of the address space of the thread from which the invocation originated. Along with the context, a *surrogate* thread is dynamically created on the remote node, and is provided with the invocation parameter page as well as selected portions of the thread's state information (i.e., the thread's *environment*). Once the surrogate thread is created at the remote node, the destination object is then mapped into its context, and the remainder of the invocation proceeds as if it were a local operation invocation. When the remote invocation completes, the surrogate thread is destroyed, changes to the thread's environment are returned to the invoking thread, and return parameters are passed back to the invoking object.

Figure 4-4 illustrates the (logical) path taken by a thread as it makes an invocation on a remote object, performs the desired operation, and then returns. In this example, **Thread**$_i$ is executing within **Object**$_a$ on **Node**$_1$, when it invokes **Operation**$_x$ on **Object**$_b$, that is currently located on **Node**$_2$. The kernel creates a surrogate for **Thread**$_i$ on **Node**$_2$, maps **Object**$_b$ into its address space, and begins execution within **Operation**$_x$. When the operation is complete, the invocation returns to **Node**$_1$ and the surrogate thread on **Node**$_2$ is destroyed.

The remote invocation service is based on a message-passing service provided by the system's communication subnetwork. In order to provide greater communications reliability than is provided by the communication subnet, the operation invocation facility requires that an invocation message be

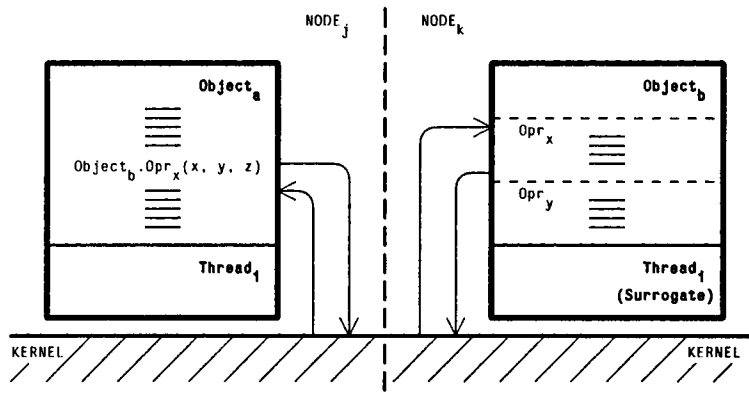

Figure 4-4: Remote Invocation

acknowledged by the recipient. This is done with timer-based protocols [Fletcher 78] that retransmit messages to the destination if no acknowledgement is received in the expected period of time. While this increases the probability that a message will get through to the destination, it also raises the possibility that more than one copy of the same message could arrive at a destination node. Thus, the remote invocation service detects duplicate messages and eliminates them within the communication subsystem, before they are passed up to the higher levels of the facility.

The Alpha kernel is (currently) not designed to cope with the forking (or the divergence) of threads. In Alpha, concurrency is derived through the use of individual threads, and the division of a single thread would introduce a significant amount of additional complexity. The forking of individual threads is therefore not supported, although new threads can be dynamically created. For this reason, it is necessary that the remote invocation service not permit failures of nodes or communication links to result in the divergence of a thread. This is dealt with in Alpha through a technique known as *active thread termination* (or *orphan detection and elimination*). If system failures cause the operation invocation facility to be unsure whether an invocation attempt succeeded, the remote invocation service ensures the termination of all parts of the thread beyond the last successful invocation.

— Atomic Transaction Invocation

If an invocation is made by a thread within an atomic transaction, the property of *thread visit notification* is added to the standard invocation service (i.e., timer-based, positive-acknowledged RPC

with retries, duplicate suppression, and orphan detection and elimination). Thread visit notification involves having the kernel track invocations made by a thread within an atomic transaction, and should a thread break while the thread is executing within the atomic transaction, all objects visited (i.e., having had operations invoked on them) as a part of that transaction are notified of the transaction's failure. Visit notification takes the form of an unsolicited invocation of the ABORT operation on the ***TransactionManager*** object instances at the affected nodes. The kernel begins tracking a thread when it invokes a BEGIN operation on the ***TransactionManager*** object, and stops tracking it when a matching END or ABORT operation is invoked on the ***TransactionManager*** object. Nesting of atomic transactions is supported by a kernel-maintained count of the depth of transaction nesting. The kernel uses the atomic transaction nesting count to determine when the outermost transaction is exited and tracking can be halted. Details of how this variety of operation invocation is used in support of atomic transactions will be covered in section 4.2.4.

— Replicated Object Invocation

The various forms of replication provided in Alpha are supported primarily through features of the invocation facility. The replication support is provided in a manner that is transparent to the client — a client performs operation invocations in the same manner regardless of whether the target object is replicated. The operation invocation facility determines whether or not the object being invoked is replicated, and carries out the appropriate operation invocation protocol for that type of object.

In the case of inclusive replication, the communication mechanism issues a message addressed to the replicated object that serves as the first phase of a two-phase invocation protocol. Each object replica that receives such a first-phase message responds either positively or negatively depending on its current state. The communication subsystem at the invoking node receives the responses and attempts to gather a minimum number of positive responses from the currently existing replicas. This minimum number of responses is specified for each operation when an inclusively replicated object is instantiated. If the initiating node does not obtain the specified minimum number of positive acknowledgements from the replicas, it issues an abort message and retries the first phase of the replicated invoke protocol. If positive acknowledgements are received from all of the replicas, the initiating communication subsystem carries out the second phase of the protocol by issuing an invocation message to all of the instances of the replicated object. In order to serialize the actions of threads within the replicas, all subsequent first phase invocations are negatively acknowledged until there are no more outstanding acknowledged invocations.

When a node's communication subsystem detects an incoming invocation from a replicated object, it waits until the invocations are received from all the necessary replicas before passing the (single) operation invocation signal to the application processor. The number of replicas involved in any replicated operation invocation is determined by the node initiating the invocation during the first phase of the protocol, and is passed to the replicas in the second phase. All invocations made by a replicated object include this count of the currently involved number of replicas that is used (in a fashion similar to that in [Cooper 84]) to merge the outgoing invocations of a replicated object.

In the case of exclusive replication, a similar type of two-phase invocation protocol is used. In the first phase the communication subsystem the node where the invocation is initiated issues a request message addressed to the replicated object. All of the currently existing replicas respond to this first-phase message and the initiating node selects one of the replicas which responded in a given period of time. The second phase of this invocation protocol involves sending an invocation message to the selected replica. The reply from such an invocation is handled in the normal manner, and outgoing invocations are also handled the same as those from non-replicated objects.

The replica selection algorithm currently in use chooses the first replica to respond to the first-phase message. This portion of the operation invocation mechanism was designed to be easily modified or replaced, in order to permit experimentation with differing replica selection algorithms. In order to use more sophisticated selection algorithms, differing types of kernel information (e.g., load statistics or resource utilization estimates) may be required in the replicas' response messages. Thus, the kernel's design permits a node's communication subsystem to have access to the kernel's internal data structures where such information can be obtained.

If more than one replica of an exclusively replicated object exists at a node at some point in time, the replicas are chained together at the same location in the node's Dictionary. Therefore, when an invocation is made on an exclusively replicated object, the first instance of any local replicas that exist on a node is used.

With both of the currently provided forms of replication, the communication subsystems use logical addressing for messages directed to all replicas of a particular object. All logically addressed messages also contain the node's physical address, and so response messages from replicas are sent using the initiating node's physical address, along with the physical address of the replica's node. Subsequent messages from the invoking node (e.g., second-phase or retry messages) are sent to the physical addresses of specific nodes to reduce the amount of logically addressed traffic on the communications subnetwork.

4.1.3. Threads

The design of threads and objects in Alpha closely mirrors the kernel's programming abstractions by, in effect, splitting typical processes into two independent components — a (passive) object part and a (active) thread part. Threads provide the components of a typical process, other than those provided by objects (i.e., the code and persistent data). This consists primarily of execution stacks and system control information.

4.1.3.1. Design

A thread in Alpha is bound to a particular context on a node, and objects that the thread performs operations on are mapped in and out of the thread's context, in the manner defined in the previous subsection. Much like normal processes, threads in Alpha can be dynamically created and deleted, scheduled for execution, and dispatched on application processing elements. Also like processes, the threads (and objects) at a particular node may require more physical memory than is available on a given node. In such a case, the representations of threads may be paged in and out from secondary storage by the virtual memory facility. Should a thread become inactive for an extended period of time, the entire representation of the thread may be swapped out to secondary storage, and brought back into primary memory when it becomes active again.

A thread exists in a region of an address space and is composed of a number of parts, each consisting of a contiguous portion of the thread's context. The major component of any thread is the portion that contains the client's stack. Threads provide each operation invocation with a separate stack that may be used to store any automatic variables declared within the scope of an operation, and is reclaimed when the invoked operation completes. Each stack is protected so that a thread can only access the variables associated with the particular operation execution underway at any point in time. Additionally, a thread has a part that contains the thread's kernel stack, and a part that contains the thread's invocation parameter pages. A thread's kernel stack is used whenever the thread enters the kernel, and is the stack on which a thread blocks (e.g., when page faults occur). The invocation parameter pages are the top two pages of the invocation parameter stack described in the previous section. The layout within a virtual address space of the various components of a thread is illustrated in Figure 4-5.

Like objects, threads have state information associated with them, analogous to process control blocks found in a conventional operating system. The Alpha kernel's thread control block includes information on the current state of the thread, an indication of the object in which the thread is currently active, the stack pointers and program counter of the thread (when it is blocked), references to the

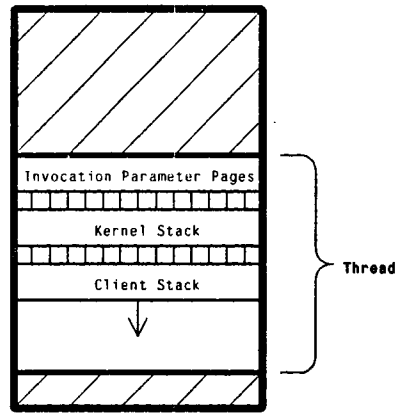

Figure 4-5: Thread Structure

thread's virtual memory data structures, and what is known as a thread's *environment.* A thread's environment contains information about the current execution state of the thread and its current attributes (e.g., resource usage, depth of atomic transaction nesting, invocation nesting depth, resource usage quotas, importance, deadlines, estimated execution times, or accumulated execution time). The environment of a thread is passed among nodes on remote invocations, and may be dynamically modified by the kernel in the course of the thread's execution or as a result of the invocation of operations on the thread.

When a thread is instantiated, a virtual memory context is allocated and bound to a newly created thread. As part of the initialization of a thread, the kernel allocates and sets up the thread's data structures, and assigns a globally unique identifier to the thread. When a new thread is created (or a thread is moved to a new node), it must be registered with the kernel by adding it to the (local node's) Dictionary. The initial object specified in the operation to create the thread must be mapped into the new thread's context, and the thread must be made ready to begin execution of the initial operation. The context and data structures created when a thread is instantiated are known (collectively) as the thread's *root.* While threads begin their existence on the node where their root is created, a thread's root may be migrated among nodes in the system (according to some higher-level policy).

Unlike normal processes, threads may extended across multiple nodes in the system. This is done when a thread makes invocations on objects that are on remote nodes, and involves the creation of surrogate threads that act as the root's agents on remote nodes. Surrogate threads inherit their root

thread's environment so as to propagate the thread's attributes across nodes (so that this information can be applied to the global management of system resources). At any point in time, a thread that extends across multiple nodes consists of a single root thread and one or more surrogate threads. Of the components that make up a thread at a given time, however, the thread is active in only one context, this is known as the *head* of the thread.

The threads that exist at a node are multiplexed onto the node's application processor. There are states associated with threads that reflect their execution state, much the same those associated with standard processes. The states that threads can be in are: RUNNING, READY, STOPPED, BLOCKED, and SLEEPING. The kernel maintains a list (known as a *Ready Queue*) at each node, containing references to the threads that are in the READY state, waiting to be multiplexed onto the processor. A thread that is bound to an application processor is in the RUNNING state, while threads that are waiting on some event are in the BLOCKED state. Threads enter the BLOCKED state when waiting for a client synchronization event, a virtual memory paging request, an I/O request, or a remote invocation. As remote invocations are made and threads move among nodes, threads on the invoking node change from the RUNNING state to the BLOCKED state. When a **DELAY** operation is invoked on a thread, the thread goes into the BLOCKED state for a specified period of time, unless the time parameter of the operation is less than a given value, in which case the thread goes into the SLEEPING state. A thread in the SLEEPING state does not relinquish the processor, but retains control of the processor until the given delay time has expired. When the **STOP** operation is invoked on a thread, the execution of the thread is suspended, just as if it were spontaneously blocked, and the thread goes into the STOPPED state. The **START** operation is used to resume the execution of threads that are in the STOPPED state. When a **START** operation is invoked on a thread, the thread is returned to its previous execution state.

The function of mapping active threads onto the system's application processors is performed for each node by the scheduling facility, supported by special hardware. Section 4.3.3 provides details of how the scheduling of threads is performed.

4.1.3.2. Rationale

The design of threads in Alpha permits an implementation that is very similar to that of standard processes, making it possible to implement the Alpha abstractions efficiently on standard (i.e., process-oriented) hardware. This design provides separate mechanisms for concurrent, asynchronous activity (i.e., threads), and the static specifications of behavior (i.e., objects). This separation of concerns aids in the construction of applications through increased intellectual manageability.

The thread mechanisms provide the functionality necessary for managing threads, without enforcing policies on the use of threads (or objects). For example, when an invocation is made to create a new thread and the designated initial object is not local to the node where the invocation is made, an error is returned. This is as opposed to the automatic instantiation of the new thread's initial object, or the automatic creation of the new thread on the node where the specified object exists, either of which imply certain object management policies.

With threads designed in this fashion, the kernel can provide a physical entity that can be strongly associated with independent, concurrent, application computations. With such a strong association it is possible for the client to provide the kernel with direct information about the application's requirements. This is as opposed to process-oriented client/server models where it is difficult for the system to determine which application-level computation a process is working on behalf of at any point in time.

In this design, a thread is associated with a client-level computation and has certain application-specific attributes associated with it (e.g., the relative importance of computations or deadlines) This allows the kernel to propagate application information throughout the system where it can be used to in making system resource allocation decisions. For example, this information is used in the global management of application processor cycles. Each node's thread scheduling facility is responsible for preempting threads running at its node when it becomes more valuable to the system for some other thread to be run. In this way, the kernel attempts to ensure that, at any point in time, the most important threads are bound to the system's application processors.

This design does not require threads to be rescheduled on each operation invocation, but rather, rescheduling is performed only when a scheduling event occurs (e.g., a more important thread needs to be run, a client synchronization primitive is used, a virtual memory request requires a delay, or a remote invocation is in progress). Also, the number of threads which can be active in an object at any one time is not restricted by the system, thus allowing a high degree of concurrency to be obtained.

4.2. Additional Mechanisms

This section describes the design of the mechanisms used to support the additional programming abstractions provided by the Alpha kernel. These are mechanisms for initialization, access control, concurrency control, atomic transactions, and object replication.

4.2.1. Initialization

The Alpha kernel's initialization is based on a common procedure to be performed for each instance of the kernel. When a node comes up (either for the first time or as a result of a restart), the standard kernel is loaded and all of the permanent objects associated with the node are reconstituted. Each node has at least one such permanent object, known as the ***Initialization*** object, that contains the a list of those objects and threads that are to be created at the node upon start-up. After reconstituting a node's permanent objects, the kernel instantiates the threads and objects contained within the node's ***Initialization*** object. The kernel instantiates entities in the order in which they were added to the ***Initialization*** object. The list of threads and objects in the ***Initialization*** object may be dynamically altered by invoking operations on it. Furthermore, the ***Initialization*** object also has the attribute of being atomically updated. This attribute helps ensure that entities can be added or removed from the ***Initialization*** object atomically and the list does not appear in an inconsistent state, regardless of when failures might occur. This initialization process is performed by an initialization thread that is created each time that power is applied to a node, and is destroyed once the node's initialization is complete.

In an embedded system, the initial set of threads and object may include all of those that the node will have throughout its lifetime. For interactive systems, however, the kernel may begin by creating a *command interpreter* object and thread with which to interact with users. This type of object may then go on to dynamically create the desired objects and threads, in response to the commands issued by the user.

4.2.2. Access Control

In the Alpha kernel, an object's protection domain is defined by the capabilities that an object possesses at any point in time. The capability mechanism in Alpha is based on the concept of system-maintained, per-object capability lists (*c-list's*). Each object has a c-list associated with it, and the manifestations of capabilities local to objects are indices into this system-protected list. Because objects are not allowed to manipulate capabilities directly, objects cannot forge capabilities and cannot pass them among themselves in an uncontrolled fashion. The list of capabilities that belong to an object at any point in time is considered an integral part of an object's state. Each object's c-list defines its protection domain, an object may only use those capabilities found in its c-list, and each invocation in Alpha involves (by definition) an interdomain jump.

There are two ways in which objects can acquire capabilities — either passed as parameters from

other objects, or provided by the kernel upon instantiation. Capabilities that are passed as invocation parameters must be identified (by the language or user) as such, so they can be translated by the system. The object making an invocation (or returning from one) places the local representation of a capability in the outgoing parameter page along with the standard variables, prior to trapping into the kernel. Following the trap, the kernel accesses the parameter page, detects the presence of capabilities, and translates and validates the capabilities. Should a capability be invalid (i.e., it doesn't exist, or it may not be passed), an error indication is returned to the invoking object, and the invocation fails. When a passed capability is validated, the system determines whether the capability should be removed from the invoking object's c-list, and removes capabilities when necessary (e.g., if the capability had *no-copy* or *no-multiple-use* restrictions). At the invoked object, the kernel adds successfully passed capabilities to the object's c-list and translates the capabilities into the object's local representation before passing them on to the object.

Capabilities that are provided to an object when it is created are called *well-known* capabilities. When defining an object type, the capabilities required by the object are specified and the kernel ensures that they are given to all instances of this type of object. The declaration of system service objects as well-known capabilities poses very little problem due to the fact that the information needed for the object descriptors is available at compile-time. In declaring arbitrary, user-defined objects as well-known capabilities, the programmer indicates a desire that instances of the specified type have a capability for an instance of an object of the well-known type.

The Alpha configuration utility ensures that the object type specification has the appropriate initial c-list, containing the descriptors for any system service objects declared to be well-known, and special descriptors containing the type identifiers of the well-known client objects. The first time an object makes use of a well-known client object capability, the kernel locates an instance of the specified type of object and replaces the placeholder in the c-list with a proper object descriptor. Subsequent invocations of operations using well-known capabilities make use of the previously obtained capability. Should no instances of the specified type of object exist, the kernel automatically instantiates an instance of an object of the specified type. Furthermore, the well-known objects are registered with the ***Initialization*** object to ensure their initial creation and regeneration following node failures.

The object descriptors associated with capabilities consist of three parts: a globally unique object identifier, a vector that indicates the operations that can be invoked on the object (both user- and system-defined), and an indication of the usage restrictions associated with each operation permitted by the capability. The defined usage restrictions are:

- **No-Transfer** — this cannot be passed as a parameter in an operation invocation
- **No-Copy** — should this be passed as a parameter in an invocation, it is removed from the invoking object's c-list (i.e., a copy is not made)
- **No-Multiple-Use** — this can only be used once and is then automatically deleted by the kernel
- **Exclusive-Use** — this can only be used by the current thread

The RESTRICT primitive is provided to the programmer to place these restrictions on capabilities before passing them as parameters in invocations. The restriction of capabilities is done on a per-operation basis within each capability and is performed by passing a (compiler-generated) mask along with the capability and operation indices to the kernel on operation invocations. It is also possible to indicate the restrictions that should be associated with the capability passed to the creator of this type of object, along with the object attributes declared in the header of object type specifications.

The protection mechanisms are dependent on efficient local validation of the capabilities used as invocation targets (and passed as parameters). There is no amplification primitive provided by the kernel, and there is no encryption on capabilities being passed on the bus. These omissions are considered reasonable optimizations in light of the system's protection goals and the nature of the intended application domain.

An alternative approach to the capability mechanism used in Alpha is an access control list technique [Levy 84]. With access lists, validation is done at the destination of an invocation, which requires the expenditure of additional communication resources and additional costs in the process of validation, beyond that required for capabilities. In fact, it is difficult to validate the rights to pass capabilities as invocation parameters within the framework of an access control list scheme. The capability mechanism provided by the kernel can be made to support access lists by making all objects well-known to all other objects, and using the kernel mechanisms that return the identifier of the invoking object or thread. Access lists can be included in objects so that they may regulate use of the resources they provide; the kernel provides capabilities and the application does the rest. This is in keeping with our belief that the kernel should manage kernel-level resources and the application should manage application-level resources (with support from kernel mechanisms).

4.2.3. Concurrency Control

In the Alpha kernel, there are two different types of concurrency control abstractions — thread mutual exclusion and data item locking. The following is a description of the mechanisms used to provide these abstractions.

4.2.3.1. Semaphores

Semaphores are associated with object control structures and therefore are moved along with objects should they move among nodes. A ***Semaphore*** object consists of a counter that contains the current state of the semaphore, and a list of references to the threads that are currently blocked on the semaphore. All operations invoked on semaphores are intercepted by the kernel, that locates the particular semaphore data structure associated with the invoking object's control structure (based on the capability of the invoked ***Semaphore*** object), and performs the desired operation on the semaphore.

In support of atomic transactions, all of an object's semaphores that a thread within a transaction has performed P operations on must have corresponding V operations issued on them should the transaction abort. This requires that the kernel be aware of all of the semaphores that a thread issued P operations on while within a particular atomic transaction. With this information it is possible for the kernel to issue the matching V operations should a transaction abort. This function is simplified in Alpha because all of the semaphores associated with an object are linked to the object's control structures and are therefore easily scanned to determine which ones require V operations to be performed on them.

4.2.3.2. Locks

The design of ***Lock*** objects is similar to that of ***Semaphore*** objects. A ***Lock*** object consists of a data structure containing an indicator of the current state of the lock (i.e., its current mode), a specification of the data region associated with the lock, and a set of references to threads that are blocked waiting to acquire the lock. As with ***Semaphore*** objects, the kernel detects all invocations on LOCK objects and performs the lock operations within the kernel. Also, the ***Lock*** object's data structures are linked with the control block of the object that created it.

To perform lock operations, the kernel determines whether the lock request is compatible with the current state of the lock, based on the kernel's lock compatibility table. If a lock request is found to be compatible, the lock is granted and the state of the ***Lock*** object is updated. If the lock request is not compatible with the current mode of the ***Lock*** object, the thread making the request is blocked

and placed into the ***Lock*** object's blocked thread list. When a lock is released, a thread in the ***Lock*** object's blocked list with a compatible lock request is granted the lock, removed from the list, and made ready to run (i.e., added to the node's Ready Queue).

In addition to their data structures, ***Lock*** objects in Alpha have special storage areas associated with them. This storage area is provided for the purpose of maintaining a type of write-ahead log in support of the functionality of atomic transactions. When a write-mode lock is acquired by a thread executing within a transaction, the kernel copies the data item associated with the ***Lock*** object into its log area. Should a transaction successfully commit, the ***Lock*** object's log can be discarded. Should the transaction abort, however, the kernel restores the data item to its previous state through the use of the lock's log.

Locks behave differently when used within transactions, however, in support of programming-level uniformity, the lock mechanism is designed to appear the same to the programmer, whether or not the threads that make use of them are currently executing atomic transactions. Locks acquired from within transactions are held beyond the point at which clients invoke UNLOCK operations, and released only when the transaction commits. Just as with ***Semaphore*** objects, all locks acquired during the execution of a transaction must be released should it abort.

The Alpha kernel makes use of locks as the primary synchronization mechanism for transactions (as opposed to optimistic [Kung 81] or timestamp-based schemes [Bernstein 81]) because other schemes rely on the notions of delays or roll-backs as a fundamental part of their normal synchronization activity. These characteristics make such concurrency control schemes undersirable for use in real-time command and control systems, where timeliness is a part of the specification of the correct behavior of the system.

In Alpha, the use of locks to access their corresponding data items is not enforced by the kernel. Without hardware support it is not practical to have the system enforce these types of locks. For this reason, the system layer must enforce the necessary discipline on the use of locks (e.g., through a language interface). While this design allows smaller data items to be locked efficiently, it also permits optimizations to be made for locking data items of page granularity (e.g., shadow-paging techniques).

A *null* lock mode is typically used as an optimization in systems where the costs of releasing and reacquiring locks is high. With locks in Alpha the more significant performance costs are associated with creating the lock object, as opposed to acquiring or releasing it. For this reason, there is no null lock mode.

4.2.4. Atomic Transactions

Most existing instances of the use of atomic transactions in operating systems evolved from the database application domain. In many such systems, atomic transactions are supported by migrating portions of a particular database system into the lower levels of the system, resulting in the propagation of a particular database model to the operating system's client. The emphasis on the design of transaction support in Alpha is directed towards more general-purpose atomic transactions. Also, the desire to make use of atomic transactions within the system layer increases the need for higher performance atomic transaction mechanisms.

The transaction approach used in Alpha was designed for flexibility and for real-time performance. To meet the flexibility goal, the transaction mechanisms in Alpha allow the client to determine when transactions are to be used and what characteristics each transaction should have. The real-time goal is addressed through a transaction management algorithm that permits bounds to be placed on the time between machine failure and subsequent transaction abort (since all affected transactions *must* abort).

An assumption made throughout this development is that the nodes behave in a *fail-stop* manner — i.e., when a failure occurs in any of a node's processing elements, the entire node is considered to have failed and a restart procedure is initiated. This assumption is made in order to avoid having to deal with issues related to Byzantine protocols [Pease 80].

In the Alpha kernel, atomic transactions are supported in part by mechanisms that directly and specifically support atomic transactions, in part by special aspects of mechanisms that are not directly related to atomic transactions, and in part by the normal aspects of kernel mechanisms.

4.2.4.1. Transaction Management Mechanisms

The primary kernel mechanism supporting atomic transactions is the ***TransactionManager*** object. An atomic transaction is initiated by the invocation of a **BEGIN** operation on the ***TransactionManager*** object by the thread wanting to initiate a transaction. An atomic transaction can come to an end in a number of ways — should all of the activities complete successfully, the thread that issued the **BEGIN** operation can issue the matching **END** operation; in the event that not all of the activities in the transaction are completed properly, an **ABORT** operation can be issued; alternatively, the kernel could detect that one (or more) of the objects (or nodes) involved in the transaction has failed, and the kernel will then issue an **ABORT** itself.

The ***TransactionManager*** object is responsible for performing the *transaction coordination* function for all of the atomic transactions in Alpha. The activity of transaction coordination involves managing the nesting of atomic transactions by individual threads, tracking all of the objects that a thread invokes operations on while within an atomic transaction, and coordinating the aborting or committing of transactions among all relevant parties. This implies that the ***TransactionManager*** object must maintain the information necessary to determine which objects are involved with each transaction. When an atomic transaction ends, the transaction manager must also ensure that all the objects involved in the transaction agree (within some period of time) to commit or to abort.

The transaction manager must also interact with other mechanisms in order to provide the various attributes of atomic transactions. For example, when a node fails, the kernel notifies the ***TransactionManager***, which then becomes responsible for invoking the **ABORT** operation on all objects that were involved in the transaction. The kernel assigns unique identifiers to each transaction; an individual atomic transaction is identified by the identifier of the thread that initiated the transaction along with the current nesting depth of the transaction.

The ***TransactionManager*** object is replicated on each node, and the replica on a particular node is responsible for those transactions that involve the objects on that node. All transaction manager replicas interact with each other to commit particular transactions, using a standard two-phase commit protocol. The **BEGIN** operation increments the transaction nesting depth count associated with the invoking thread, and causes the invocation mechanism to begin tracking the invocations made by the thread. The **END** and **ABORT** operations cause the ***TransactionManager*** object to discard information about a particular atomic transaction and interact with all of its instances to ensure unanimity in committing or aborting the atomic transaction. The ***TransactionManager*** object invokes the transaction-related standard operations on the objects involved in a transaction on a commit or an abort. Furthermore, when an **ABORT** operation is invoked on the ***TransactionManager***, control is returned to the statement in following the **END** operation for the transaction being aborted.

When transactions are aborted at the explicit command of the thread and there are no node failures, the ***TransactionManager*** object is able to use the information it has to determine which objects are involved in the transaction and notify all of them of the transaction abort. When a transaction abort is brought about by the failure of nodes, however, a portion of the transaction manager's information is lost. The particular node at which a transaction was initiated (and therefore the node whose ***TransactionManager*** object replica serves as the site coordinator for the transaction) may be lost in a failure. Thus, the notification of affected objects must be performed by a decentralized transaction

coordination algorithm. The complete abortion of a transaction is necessary to satisfy serializability constraints and to allow failed computations to be "undone." Therefore it is important that the transaction management algorithm guarantee the complete and timely abortion of failed transactions.

To satisfy these requirements, the kernel's transaction abort management is based on the use of a technique known as a *dead-man switch*, whereby each node autonomously manages (by way of its ***TransactionManager*** object replica) the transactions that involve objects existing local to the node. In this approach, an *abort time* is assigned to each transaction as it is created. This time is never later than the current time by more than the *abort interval*, thus bounding the maximum time between a failure and the completion of abort. Periodically, the node at which the transaction was initiated executes a two-phase refresh protocol with all of the nodes on which objects involved with the transaction exist, assigning a new abort time to the transaction. If the refresh protocol does not complete successfully (i.e., some node cannot respond), all of the nodes involved in the transaction being refreshed are autonomously aborted at their given abort time. The abort interval may be adjusted by trading a shorter interval for higher communication overhead and processing overhead.

This design of the ***TransactionManager*** object has several benefits. The two most significant are that it bounds the abort interval, and that it correctly handles orphans. A proof of this algorithm appears in [McKendry 85], along with a more detailed discussion of this algorithm. This design also permits optimizations that may significantly reduce the required message traffic, although in its current form the algorithm requires message traffic proportional to the square of the number of nodes in the system. The actual amount of message traffic in any implementation can be chosen according to the desired abort interval.

4.2.4.2. General Mechanisms

In addition to the kernel mechanisms directly intended for the support of atomic transactions, a number of other kernel mechanisms provide support for threads executing within transactions. Certain kernel mechanisms behave differently (i.e., take on special characteristics) when the thread making use of them is inside an atomic transaction, while the normal characteristics of other mechanisms are used to support atomic transactions. Mechanisms that have special features (or behaviors) to support atomic transactions include the operation invocation mechanism and the thread synchronization mechanisms. The permanence and atomic update attributes of objects, the standard operations associated with client objects, and the thread repair feature of the operation invocation mechanism all contribute to support atomic transactions in Alpha.

— Operation Invocation Facility

The operation invocation facility's *visit notification* feature supports the property of transaction atomicity — all of the actions performed within an atomic transaction are aborted if any one of the actions within the atomic transaction fails. The operation invocation facility detects whether a thread is in a transaction, and if so, maintains records of all of the objects that the thread visits while in a given transaction, for the ***TransactionManager*** object's use. Additionally, the *thread repair* aspect of the kernel's operation invocation facility is a standard feature that ensures orphans are detected and eliminated. The property of orphan elimination also supports the serializability attribute of atomic transactions. Both of these features of the operation invocation facility are based on the fact that the communication subsystem detects node failures and reports them to the ***TransactionManager*** object. This is done by assigning a watchdog timer to each thread that arrives at a node, and periodically refreshing the timer. When a thread's timer expires, the thread is deleted and the ***TransactionManager*** object is notified. Each transaction's coordination site is responsible for refreshing the timers at each node where the transaction exists. This refresh protocol forms the basis for the transaction abort scheme described above.

— Thread Synchronization Mechanisms

In Alpha, all of the synchronization mechanisms are based on suspension of the logical progress of threads at specific points within objects. To support atomic transactions, these mechanisms allow the operations performed on them by threads within atomic transactions to be undone if the transaction aborts.

In order to support the serializability constraints of atomic transactions, a two-phase locking discipline is enforced on locks acquired within atomic transactions in Alpha. When a thread executing within an atomic transaction invokes the **UNLOCK** operation on a ***Lock*** object, the kernel recognizes that the thread is in a transaction and does not actually release the lock until a **COMMIT** or **ABORT** operation is invoked on the object (which occurs when either the thread commits its top-most atomic transaction, or when the atomic transaction aborts).

The fact that the Alpha kernel uses locking as a basic mechanism for atomic transactions raises the question of lock granularity. There is some debate over whether or not page-level locking is adequate for practical systems [Traiger 82]. It seems clear, however, that implicit locking does not provide a very high degree of concurrency when control and data information are mixed in a memory page. In the Alpha kernel there are optimizations for locking page-sized data, in addition to the write-ahead

logs used for locking smaller-sized data items. Because Alpha is a virtual memory-based system, consideration must be given to the interaction between virtual memory paging activity and the activities associated with the permanence and failure atomicity attributes of atomic transactions. For example, if paging is done to an object's secondary storage image, the secondary storage image of the object may be in an inconsistent state after a node failure (in the sense that the image may contain data that has been modified by uncommitted atomic transactions). In order to deal with this, the normal paging activity is suspended (for the pages affected) while an object is being manipulated by a thread within an atomic transaction.

— Standard Operations and Object Attributes

A number of standard operations defined on objects by the kernel serve to support atomic transactions, and the optional attributes of objects also play a significant role in providing the characteristics of atomic transactions. The standard **UPDATE** operation interacts with the locking mechanism in order to ensure that only the consistent state of objects is used in the update activities. This is to permit the visibility requirements of atomic transactions to be met, and to provide the "undo" function needed by atomic transactions. The **UPDATE** operation is used to write the current state of the object to its secondary storage image. Only the committed portions of an object are written to the secondary storage image. This means that if any locks are held on the object, the committed data stored in the locks' write-ahead log is used in place of that which is currently in the primary memory image of the object. Depending on the attributes of the object in question, this **UPDATE** operation may have permanent effects, may occur atomically with respect to node failures, or may have a non-atomic or transient effect.

Additionally, the kernel provides objects with default standard operations related to atomic transactions, (i.e., the **PRE_COMMIT**, **COMMIT**, and **ABORT** operations). The default operations to support atomic transactions can be replaced with specialized operations by the client. These mechanisms ensure the proper recovery of all objects following a node failure. This is done by aborting all changes, should failures occur prior to the commit point, and by repeatedly attempting to complete the transaction once the commit point has been reached (provided that all of the completion operations are idempotent).

Non-serializable transactions are supported in Alpha by relaxing some of the constraints the kernel places on the behavior of threads operating within transactions (e.g., by releasing locks immediately) and allowing the client to provide his own transaction-related operations for each object. In this way,

specialized knowledge of the applications and objects can be applied by the client to improve overall system performance.

In order to permit the use of objects by threads both within and outside of atomic transactions, a technique is used that is similar to that used for locking within the Locus system [Popek 81]. All of the data items associated with locks are handled the same with respect to **UPDATE** operations, regardless of the locking mode. That is, data items are committed (i.e., written to the secondary storage image) once the outermost transaction commits — even those data items whose locks were held only in read-only mode.

4.2.5. Object Replication

The replication schemes currently supported by the kernel are based on a major underlying assumption that the partitioning of system resources (as it is popularly conceived of in the literature [Herlihy 85]) do not occur in the Alpha environment. This derives from the fact that systems in the real-time command and control application domain are typically provided with sufficient physical redundancy to ensure that the bisection of the distributed computer's interconnection subnetwork could only result from a catastrophic failure of the platform in which it is embedded. In effect, the communication interconnection subnetwork in Alpha is considered equivalent to a backplane in a conventional uniprocessor, and similar failure assumptions are made.

In Alpha, the client creates a replicated object by invoking the **REPLICATED_CREATE** operation on the ***ObjectManager*** object. This operation instantiates a specified number of instances of a given object type, each of which share a common object identifier. The multiple instances of a replicated object project to the client a view of a logical object that has different levels of availability, responsiveness, and consistency, than corresponding non-replicated objects.

Each of the individual instances that comprise a replicated object appear similar to all other objects, with the exception of the fact that they all participate (to some degree or other) in operations invoked on the replicated object. An object invoking an operation on the replicated object does not need to be aware of the fact that the target object is replicated; invocations on replicated object appear identical to the invocation of a non-replicated object. Because the support for replication is provided by the invocation facility, functions involved in the management of replicated objects (e.g., quorums or multiple requests and responses) are performed within each node's communication subsystem and therefore do not incur significant performance penalties on the application.

Details of the replication schemes supported in Alpha are described in section 4.1.2.2. While the basic mechanisms for replicated object support are provided by the kernel's invocation mechanism, the remainder of the replicated object management policies are performed by objects in the higher layers of the system. The replication mechanisms currently supported in Alpha are representative of a broader collection of replication mechanisms that support a greater range of replication policies.

4.3. Major Facilities

This section describes the design of the major facilities in the Alpha kernel. These facilities do not directly correspond to specific programming abstractions, but provide support to many of the kernel's mechanisms.

4.3.1. Inter-node Communications

In Alpha, the remote operation invocation mechanism is supported by the inter-node communication subsystem. The communication subsystem employs an additional point of hardware control, a programmable communications subnetwork interface, so that more complex functionality can be performed by the communication facility without incurring additional application processing overhead. The majority of the inter-node communications functionality was designed to be executed on the communications processing element, with only a minimal amount of functionality remaining within the kernel proper (i.e., executing on the applications processing element). The communications processing element is one of the functional units that make up a node in the Archons project testbed (see Chapter 6). The communications processing element shares a portion of the node's memory with the application processing element, and interacts with it through the node's interprocessor interrupt structure.

The communication subsystem provides the kernel with a high-level interface to the communication subnet. When the kernel wishes to make use of the mechanisms provided by the communication facility, it places a command block in the shared memory region and interrupts the communication processor, and the communication subsystem interacts with the application processing element in a similar fashion. The result of this design is that much of the communication overhead typically associated with distributed systems is handled by the communication subsystems and the application processors are only interrupted to signal high-level (asynchronous) communication events — e.g., when a complete message has been received, manipulated, and loaded into the address space of the destination object.

The functionality provided by the communications facility may be quite involved, requiring non-trivial amounts of processing power and the multiple exchanges of packets. For example, the communication facility handles all of the low-level aspects of the operation invocation function. This includes the handling and generation of positive and negative acknowledgments, communication timeouts, and packet retransmissions, in addition to such typical communication functions as message disassembly/reassembly, encapsulation, and page alignment of message data. The communication facility is responsible for executing an inter-node liveliness protocol. This protocol requires that each node monitor the transmissions of the other nodes, and send explicit queries to nodes that have not been heard from in some period of time. This is done in order to determine which nodes are working and which are not, and to provide timely support to such invocation functions as thread repair and visit notification. Also, the communication facility includes mechanisms in support of specialized protocols for such functions as logical clock synchronization, two-phase commit protocols, the bidding protocols used to deal with a replicated set of objects, initial program loading, and node reboot functions. In addition to these protocols, the communication facility contains a mechanism that permits a remote node to gain access to the application processor and local memory in a node to perform remote test and diagnostic procedures. This mechanism also supports the remote loading and storing of a node's primary memory for restart, fault location, and debugging.

Among the design goals of the communication facility were that it be flexible and extensible, without excessive performance penalties. The communication facility was designed to allow the addition of new protocols (or the modification of existing ones) with a minimum amount of time and effort. It was also designed to make it possible to increase the performance of the communication facility as transparently as possible, through the inclusion of additional points of hardware control (i.e., additional hardware functional units). Also, the communications subsystem has an internal, low-level interface that facilitates the interchange of link-level interconnect structures and supports multiple (replicated) interconnects.

It has been repeatedly shown that a large proportion of the cost associated with message-passing systems is related to the movement of messages among address spaces [Jensen 78a, Wittie 79, Fitzgerald 85]. To deal with this problem, the communications processor and the application processor interact through a shared virtual memory interface, sharing memory by the manipulation of virtual memory mappings. This provides a significant performance advantage over the copying of blocks of memory, as is typically done in message-passing systems. The communications subsystem moves information to and from the communication subnetwork without copying it among intermediate-level buffers. This requires that the communication subnetwork interface hardware

deposit the information it receives directly into the physical memory location where it will ultimately reside, and remove the information it must transmit directly from the location in which it was placed by the applications processor. If this is the case, then all other movement of the communicated data are performed via memory mapping operations.

The location-transparent access to objects provided by the operation invocation facility is based on the use of global object identifiers and is supported by the communication facility through the use of logical destination addressing. For each operation invocation, the kernel performs a lookup operation on the node's Dictionary to determine whether the destination object is local, and if not, the invocation request is passed to the communication facility. The communication facility addresses all remote communications to destination objects, using their global identifiers. The communication facility's address recognition mechanism uses the local Dictionary to determine which objects are local to the node, and therefore which (logically addressed) packets should be received. This mechanism supports object-level dynamic reconfiguration by simplifying the location of objects — i.e., the maintenance of logical to physical address translation tables is not necessary. In addition, the use of logical addressing also supports the use of replicated objects, by having more than one object sharing a logical name.

4.3.2. Virtual Memory Management

In Alpha, objects and threads have representations that exist in their node's primary memory. It is the responsibility of the virtual memory facility to manage threads' virtual address spaces, the primary memory images of objects and threads, the primary memory image of the kernel itself, and the interface between primary and secondary memory. Whenever an instance of the kernel is initialized at a node, an object is instantiated, or a (root or surrogate) thread is created, the virtual memory mechanisms are used to create and initialize the virtual memory data structures for the instantiated entity. The kernel provides mechanisms for removing these structures when objects or threads are deleted.

When the local scheduling facility decides that another thread should be bound to its application processing element, the currently running thread must be suspended so that execution can begin on the other. This activity requires a context swap, which involves the modification of the address translation unit entries of the application processor, in order to provide the correct virtual to physical address mapping for each thread's context. As operations are invoked, objects are mapped in and out of the invoking thread's context. This amounts to a *partial* context swap, where only the mappings

for the object portion of a thread's context are modified. The virtual memory facility provides mechanisms to perform both full and partial context swaps.

Objects exist in a node's primary memory, and may be mapped into any number (including none) of the contexts at a given time. The virtual memory management facility is responsible for keeping track of the physical location of objects and managing their mappings so that all of the objects on a node can be located and mapped into a thread's context when operations are invoked on them. Also, the virtual memory facility ensures that the physical memory image of objects that are simultaneously mapped into multiple contexts can be shared.

The current kernel design has the kernel mapped into a portion of each thread's context. Only a portion of the kernel must be locked in memory, whereas much of the kernel is pageable, just as are objects and threads.

All of the major entities (threads, objects, and kernel instances) have primary memory representations that are composed of a collection of memory *Extents*. An Extent is defined to be a contiguous region of memory that has a specific function associated with it, as well as a particular set of common characteristics (e.g., read/write/execute protection mode). The *code*, *data*, and *heap* portions of an object are each examples of Extents, while the entire primary memory representation of a thread (with the exception of its kernel-maintained control structures) is also an Extent. Extents are logical entities that are independent of the physical structure of either the primary or secondary memory. The virtual memory mechanisms map Extents onto whatever physical structures the underlying hardware provides.

Each major entity in the Alpha kernel has a data structure known as a *control block* associated with it, and each of the entities that make use of secondary storage has references to virtual memory data structures in its control block. In Alpha, every kernel entity contains all of the virtual memory information related to that entity. In particular, for any Extent there is a data structure (known as an *Extent Descriptor*) that contains all of the necessary virtual memory information for that portion of memory. An Extent Descriptor contains information about the current state of the Extent, along with the identifier(s) of the type or instance secondary storage object(s) with which the Extent is associated. Also, each of the entity's control blocks contains structures that indicate the context(s) it is currently bound to (if any). This design distributes the virtual memory information for each entity, as opposed to centralizing it in a global system mapping table (as is required by some memory management hardware). This design was chosen because it is modular and independent of any particular

memory management hardware, thus it helps to isolate the virtual memory facility from machine dependencies. Should certain hardware require a particular data structure for the system's address translation unit, such a structure can easily be created from the information distributed among each of the entity's data structures.

In the current testbed hardware, the application processor's memory management unit consists of a two-layer translation table (known as the segment and page tables). The kernel manages these tables as an address translation cache, with the complete address translation structures in primary memory (as a part of the data structures associated with each kernel entity). Also, parallel data structures are provided by the kernel to augment the translation tables so the kernel can more easily identify the entity with which the particular memory translation table entry is currently associated. As a further optimization (trading off space for speed), a data structure is maintained by the virtual memory facility that indicates which entity each physical memory page is currently associated with. Using these structures, the entities to which various memory resources belong can be quickly identified when a virtual memory operation is to be performed (e.g., when a page fault occurs), thereby enhancing the performance of the virtual memory facility.

4.3.3. Application Processor Scheduling

The Ready Queue for each node is maintained within, and managed by, the node's scheduling subsystem. This permits the computation of thread execution schedules for the node concurrently with the execution of the applications processing element.

Threads are added to a node's Ready Queue when new threads (roots or surrogates) are created, or blocked threads become ready, and threads are removed from their Ready Queue whenever they leave the READY state (i.e., when they are completed, stopped, blocked or deleted). When a thread is added to the Ready Queue, the thread's environment information is passed to the scheduling subsystem to provide the necessary inputs to the scheduling algorithm. When changes are made to the environment of a thread that is currently in a node's Ready Queue, the modified information must also be passed on to the scheduling subsystem. Because the environment information moves with threads among nodes, scheduling decisions are made based on the global attributes of computations, as opposed to purely the local scheduling decisions made in less tightly-integrated systems.

The scheduling subsystem continually examines the information associated with the threads in the Ready Queue, and orders these threads according to the relative value to the system of their completion, as defined by the given scheduling policy. When it becomes more valuable to the application

processor to execute a thread other than the one currently bound to a node's application processor, the scheduling subsystem preempts the currently executing thread and starts the new thread. In Alpha, the binding of threads to the application processor is performed when scheduling events occur (e.g., when it becomes more valuable to the system to run another thread or when the currently executing thread blocks). This is different from the more traditional approach in which processes are rescheduled based on events only marginally related to the scheduling policies (e.g., as a side-effect of communications or following some periodic time-slice interrupt).

The design of the scheduling subsystem is such that a wide range of different scheduling policies can be specified with little effort. The policy in effect can be selected dynamically by the application, or bound at system configuration time. Currently the kernel uses a deadline scheduling algorithm when the computational demands on an application processor can all be met. When the demands for computational resources cannot be met, the threads are scheduled in such a fashion as to maximize the value to the system of the computations that can be performed. The currently used overload handling heuristic is known as a *best-effort* approach, and is similar to the technique described in [Locke 86]. This scheduling policy has been simulated extensively, and promises to perform well in both the normal and overload cases.

To perform the desired scheduling policy, the scheduling subsystem requires certain information about each thread in the Ready Queue. The information provided by each thread's environment currently includes an indication of the estimated amount of processing time that the thread needs across a specified interval of time. The scheduling subsystem monitors the amount of processing time that each thread has accumulated during the current interval, together with the thread's environment information, and generates a deadline schedule for all of the threads in a Ready Queue. To handle the overload case, each thread's environment also includes a user-specified indication of the thread's current, global relative importance (i.e., a partial ordering).

The client uses a block-structured construct to specify the deadlines for regions of code. A qualifier is used with the deadline construct to indicate the type of deadline. Currently, there are two types of deadlines that can be specified — *hard* deadlines, that indicate the computation has no value if its deadline cannot be met, and *soft* deadlines, that indicate some relative value is associated with the completed execution of this computation even if its deadline cannot be met. When a node experiences a computational overload, the deadlines of some threads may not be met. To minimize the waste of system resources, the system aborts some threads from the Ready Queue. The same mechanism is used for this as is used for aborting threads in atomic transactions or aborting broken

threads. Control is returned to the point in an object following the end of the deadline block, and the status primitive indicates a failure. Because deadlines can be nested like transactions, a similar collection of information must be maintained by the kernel to keep track of which deadline is currently active.

4.3.4. Secondary Storage

In the Alpha kernel a number of entities require physical memory, including threads, objects, the kernel itself, and the data structures to support each of them. Practicality frequently limits the amount of memory available at a node so that not all of these entities can reside in a node's primary memory at once. Furthermore in traditional systems, a node's primary memory is volatile with respect to node failures. To deal with these issues, the Alpha kernel provides a secondary storage facility that maintains images of the primary memory entities. However, the fact that images of dynamically changing primary memory entities must be kept in secondary storage introduces problems related to update consistency that must also be dealt with.

In the design of Alpha, an attempt was made to conceptually unify primary and secondary storage, and to provide more direct support for the functionality desired by the users of the interface between the two forms of storage. The interface provided to the secondary storage facility allows the virtual memory mechanisms to indicate their (logical) desires to the secondary storage facility, which then is is responsible for executing them. For example, when a physical memory page is selected as a victim by the page replacement mechanism, the page number is provided to the secondary storage facility along with the request to page-out the victim, or when a page fault occurs, the fault handler submits the page number along with a page-in request. The secondary storage facility accepts such requests and independently determines their proper disposition. For page-out requests, the secondary storage facility determines whether the page should actually be written to secondary storage (or whether an identical copy of it is already in secondary storage), while for page-in requests, the secondary storage facility determines the location in secondary storage from which the requested page is to be retrieved. This is as opposed to the more traditional, low-level, open/close/read/write interface to secondary storage. In Alpha, the secondary storage facility abstracts out all of the low-level details of the secondary storage hardware, and provides high-level support for the desired kernel abstractions. For example, issues of what kind, how much, and where in the system secondary storage physically exists are not made visible to the user of the secondary storage facility in Alpha.

The top-down design of the secondary storage facility involved an examination of the functional

requirements derived from the specification of the kernel's programming abstractions, and resulted in the generation of a number of non-traditional, orthogonal mechanisms. Each of the resulting mechanisms provides an individual function that can be combined with other of the facility's mechanisms in various ways to provide the functionality desired from secondary storage. The separation of concerns afforded by this mechanism-driven approach allows each of the mechanisms to be optimized to better provide the desired services without sacrificing the flexibility of the facility.

The objects maintained within the secondary storage facility are registered in the kernel's Dictionary, and may be accessed through the operation invocation facility in the same manner as objects in primary memory. Therefore, the same functionality afforded to client objects by the invocation facility is available to objects in secondary storage. The use of the operation invocation mechanism as the interface between primary and secondary storage supports the construction of a reliable and available secondary storage facility, with location-transparent global access to the objects maintained in secondary storage.

The secondary storage subsystem can derive a great deal of benefit from the use of an additional point of hardware control at each node with secondary storage resources. By providing the secondary storage facility with independent processing support, it becomes possible to exploit the concurrency available between the application processor and the secondary storage device(s), both at a given node and among nodes. Additional concurrency provided for this facility also makes it possible to perform a number of optimizations that involve the use of various types of secondary storage technologies. For example, bulk semiconductor memory may be used for caching, for transient object storage, and for object paging, in addition to conventional forms of non-volatile storage (e.g., magnetic medium storage) for permanent objects. The addition of computational power to the secondary storage subsystem can also be applied towards the support of highly available object storage and high-performance, atomically updateable secondary storage.

The secondary storage facility maintains two different kinds of entities — *type* objects and *instance* objects. When an object is instantiated, the secondary storage facility creates an image of the object in secondary storage known as the instance object. When an object is instantiated the "type" of the object must be specified. This "type" identifier refers to a specific object template, maintained in secondary storage, known as a type object. The secondary storage facility maintains the mapping between the logical identifier that is used by clients to specify object types, and the type object, which is maintained in secondary storage. In Alpha, all threads, objects, and even the kernel itself have type and instance objects associated with them. Also, instance objects may contain all of the control

information associated with the kernel entity (e.g., control blocks or parameter pages) in addition to information that is mapped into contexts. This is useful in checkpointing and migrating entire threads or objects. Both type and instance objects are registered in the Dictionary, to allow access via the invocation facility.

The secondary storage facility supports the demand-paging virtual memory support offered by the Alpha kernel. With the exception of the kernel's page fault handling code and data, all of a node's primary memory may be paged. Paging is performed in a fairly conventional manner — page replacement is currently done according to a simple FIFO-like algorithm. Each entity in the kernel (i.e., threads, objects, and the kernel itself) is paged to its instance object. The virtual memory facility maintains the mapping between objects in primary memory and their instance objects, while the secondary storage facility maintains the mapping between each instance object and the type object it is associated with. The secondary storage facility uses this information to intelligently satisfy paging requests. For example, the secondary storage facility recognizes the fact that information that is already in secondary storage (e.g., code and unmodified pages) need not be written to satisfy a page-out request.

The secondary storage facility is also responsible for providing the permanence and atomic update attributes of objects. If a client object has the attribute of permanence, its instance object is kept in a non-volatile portion of secondary storage. While secondary storage is necessary to support the virtual memory requirements of the Alpha kernel, it is not required to be non-volatile. The fact that all objects in Alpha are associated with type and instance objects within secondary storage makes possible the paging, swapping, migration, and permanence of objects. The fact that part (if not all) of the secondary storage in Alpha is non-volatile makes possible the various degrees of object persistence across machine outages and other system failures. The secondary storage facility also provides the attribute of atomically updated objects. This ensures that either the entire update of an object is done correctly, or no changes are made (i.e., the secondary storage image of the instance object remains in its previous state). When the kernel wishes to checkpoint an object, it uses the secondary storage facility's *update* mechanism, which determines what needs to be written, and makes the changes atomically with respect to failures and external visibility. This mechanism is part of the high-level interface provided by the secondary storage facility, and is used in the UPDATE operation defined by the kernel on all objects. Both of these properties of objects — permanence and atomic update — are specified when an object is instantiated and can be dynamically altered in the course of an object's lifetime.

4.4. Optimizations

This section describes the design optimizations which, while not strictly necessary for the logical function of the system, are included to improve the overall system performance.

4.4.1. Client/Kernel Objects and Threads

As an optimization, an attribute of objects is provided that determines whether it is a *client object* or a *kernel object*, i.e., whether the object exists in a separate virtual address space or within the kernel's context. This optimization allows the system-builder to migrate a limited number of threads and objects downward into the kernel to achieve an increase in their performance.

This kernel object attribute is specified by the client with a compile-time declaration, placed in an (otherwise) standard object specification. Kernel objects are instantiated by a passing the type identifier for the desired kernel object along with the CREATE operation invoked on the ***ObjectManager*** object. This optimization can be performed on any object provided that there is sufficient space in the kernel's context to contain it.

To a thread making an invocation on an object, there is no distinction between invocations of operations on client or kernel objects, however the kernel's invocation mechanism deals with each of these cases quite differently. Furthermore, an invoked kernel or client object is not aware of whether the object that invoked it is itself a kernel or a client object.

Just as the client/kernel optimization exists for objects, there is also such an optimization associated with threads. In Alpha, a client thread is bound to its own virtual address space, separate from that of the kernel, and when it invokes operations, client objects are mapped in and out of the thread's context. Should such a separate context not be needed, a *kernel thread* can be used. Kernel threads do not have their own context, but rather execute entirely within the kernel. Whereas a client thread and object appear as a traditional process with its own virtual address space, a kernel thread appears as a simplified form of a process that shares the kernel's context.

Kernel objects may invoke operations on both kernel and client objects. Kernel threads may be explicitly created by passing the appropriate parameter along with the CREATE operation invoked on the ***ThreadManager*** object. Kernel threads are placed into the scheduling mix and are multiplexed onto the application processors in the same fashion as client threads.

These kernel optimizations are provided primarily for the purpose of adding functionality to the kernel. The intent is to allow the kernel builder to develop sophisticated kernel functions as client objects, and when the functions are completed and debugged, they can be migrated into the kernel to achieve a higher degree of performance, simply by altering the object's attributes declaration and recompiling. Additionally, kernel threads provide a form of light-weight processes that can be used to obtain low-cost concurrency within the kernel.

The increase in system performance provided by kernel objects derives from the elimination of the virtual memory manipulations and partial context swapping associated with invocations on client objects. Because kernel objects share the kernel's virtual memory control structures there are fewer data structures associated with kernel objects, thereby making them somewhat more space efficient.

In addition to the increased performance provided by kernel objects, there are a number of other side-effects that must be considered. Kernel objects must co-exist in the kernel's context with the kernel and all of the other kernel objects, thus sacrificing the fault containment properties afforded a client object. Because kernel objects are bound to the kernel at link-time, it is not possible for objects to change an object type's kernel/client attribute at run time. For the same reason, it is not possible for specific instance of kernel objects to dynamically migrate among nodes in the system.

Because the fault containment provided by placing objects and threads into different virtual memory domains has been sacrificed for performance, kernel objects must be trusted objects. This is because the failure of a single instance of a kernel object can cause the entire kernel to fail. Because kernel objects are linked into the kernel itself, they execute in supervisor mode and may reference code and data that is not logically a part of the kernel object.

4.4.2. Kernel Services

All kernel services are provided through object interfaces; these objects are known as *system service objects*. The operations provided by system service objects may be viewed as a collection of entry points into the kernel (i.e., system calls). While these objects appear to be the same as the client objects that make use of them, they are in fact implemented in such a manner as to provide a high degree of performance enhancement over actual client object implementations.

All invocations of operations on system service objects are handled by the invocation mechanism. Also, the invocation mechanism assumes (unless explicitly indicated otherwise) that the system service object local to the invoking object is to be used. In the case where an operation is invoked on a

local system service object, the invocations are optimized to identify the system service object to be used, locate the appropriate entry point within the kernel, prepare the parameters, and begin execution of the kernel code that implements the desired operation. System service objects are designed as simple subroutines that are compiled into the kernel, and are located through a simple lookup mechanism that maps system service object and operation identifiers to kernel entry points.

4.4.3. Critical Resource Preallocation

The overall performance of the Alpha kernel is highly dependent on the cost of dynamically allocating certain critical resources. These resources are physical memory pages, kernel thread data structures, and client thread data structures. In this approach, anonymous, free instances of the resources are maintained in pools, each managed by a kernel daemon (i.e., a kernel thread and a kernel object) whose task it is to ensure that the level of its pool stays within the specified limits.

It is the responsibility of the daemons to ensure that the number of resources in the various pools remain withing their specified limits. High, normal, and low limits are established for each pool. Whenever resources are removed from, or returned to a pool, its level is examined to determine whether it is within the given limits. If not, the pool's manager daemon is unblocked and attempts to return the pool to its normal level. Once unblocked, a daemon remains active until the level of the pool is within some offset of its specified nominal level, thereby providing a form of hysteresis to avoid stability problems.

The effect of making the high and low pool limits too close together may be poor system performance because of the additional processing overhead incurred by the frequent activation of the resource management daemons. If the limits are too far apart, however, memory resources may be wasted to keep an excessive number of free resources in the pool, also resulting in the degradation of system performance. The choice of pool limits is therefore an important decision that is best tuned for each particular system configuration. Alpha's pool limits are currently static, however later implementations could dynamically alter them based on dynamic measures of system run-time behavior. The ideal case is where there is always just one resource in the pool, i.e., there is a minimum number of resources tied up in the pool, and there is a minimal amount of waiting when an object requests a resource. In order to approximate this, some degree of analysis and tuning must be done to choose the appropriate values for the limits on the various pools.

In order to utilize the local system resources in Alpha effectively, the pool limits are set to minimize the number of resources that are in the pool, while attempting to always have one resource available

when a request arrives. In the event that a request is made for a resource and the pool is exhausted, instead of blocking the requester until the resource manager daemon can run and replenish the pool, the allocation routines are designed to allocate a resource dynamically. This suggests that most of the time resources are provided quickly in response to a request, but under high-demand conditions the pool may be depleted, in which case the full cost of acquiring the desired resource must be paid.

The intent of this resource management strategy is to pay the cost of resource allocation at a time when the system can best afford it (i.e., during low load conditions). During peaks of higher system loading requests for resources are satisfied from a preallocated pool. This is similar to how buffering is used in some queuing problems — it does not deal with the fact that a sustained incoming rate in excess of the outgoing rate will result in overflow, but rather smooth out momentary peaks in the rate of flow.

A drawback of this approach, however, is that a daemon must execute in order to allocate resources. Under conditions of high demand for these critical resources, it is typically the case that there is also an accompanying peak in demand for processing cycles. Therefore, this resource management strategy could add to the system's processing, at one of the worst possible times. This problem is dealt with by both tuning the pool limits, and by degenerating to on-demand allocation when a resource pool is exhausted. Therefore, in the worst case, all of the critical resource pools will be empty and the allocation routines will be allocating resources completely on-demand.

5
Kernel Detailed Design

This chapter provides an overview of the detailed design of the Alpha kernel. It starts with a description of the kernel's implementation structure, then provides details related to the major internal data structures, followed by details of the kernel mechanisms.

5.1. Implementation Structure

In its implementation, the Alpha kernel assumes the logical system structure shown in Figure 5-1. This structure defines several intermediate levels of abstraction between the hardware and the application, each of which will be explained in the following text.

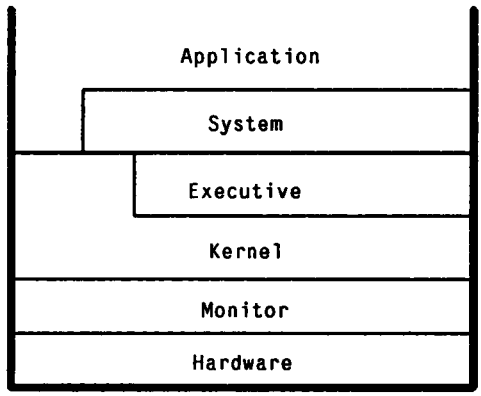

Figure 5-1: Logical View of the Implementation Structure

5.1.1. Application and System

These levels are both composed of client objects, and the only difference between the application and system layers is that objects in the system layer are considered to be trusted, and therefore may be given special privileges (provided and enforced through the use of capabilities). In embedded systems, however, even this distinction between application and system layers may not hold. Nevertheless, these layers are the principle clients of the Alpha kernel and serve as the ultimate test of the kernel's effectiveness.

5.1.2. Executive

The executive layer employs kernel objects and kernel threads to augment the basic system functions provided by the kernel. This layer unifies the local agents contained in the individual kernel instances under a single, system-wide resource management policy. Examples of the threads and objects that might exist in the executive layer are: the idle object, the secondary storage manager, the work assignment manager, the replication manager, the paging manager, and the swapping manager. Furthermore, all of the objects in the executive are (by definition) kernel objects, and similarly all of the threads are kernel threads.

5.1.3. Kernel

The Alpha kernel layer supports the previously defined programming abstractions and system interface. The kernel is composed primarily of C language routines, with a few assembly language routines. The kernel routines are divided into two groups: *system service objects*, which are groups of routines that present an object interface to the clients of the kernel, and basic support routines, which provide services that support the kernel abstractions, but are not necessarily visible to the kernel's clients.

The kernel routines are grouped according to their logical function, corresponding to the *facilities* provided by the Alpha kernel. Each kernel facility is composed of one or more *mechanisms* that perform a specific aspect of a facility's functionality. The Alpha kernel facilities include: operation invocation, object management, thread management, concurrency control, replication management, and transaction management. With the exception of the operation invocation facility (whose function spans all of the nodes in the system), the kernel mechanisms perform their functions local to the node on which they exist. Each of these mechanisms provide their particular service to the kernel's clients via an object interface, and all access to kernel-provided functionality is achieved via the operation invocation facility.

For the most part, the kernel mechanisms are implemented as (one or more) routines that make use of specific hardware components at a node. In support of the kernel mechanisms, a number of lower-level mechanisms are provided as routines that manage internal data structures, primary memory, inter-node communications, secondary storage, and peripheral devices. The remainder of this chapter is dedicated to describing the implementation of the kernel mechanisms.

5.1.4. Monitor

The monitor is the lowest level of software in the system, and it provides low-level support for managing the underlying hardware at each node. The monitor is a collection of routines that reside in the on-board PROMs of each processor in a node. The monitor provides both interactive support for test and debug via the console, as well as general-purpose run-time support for the kernel. Among the functions provided by the monitor are routines for:

- performing power-on reset diagnostics and initialization of all of the hardware at a node (including the processor);
- managing the low-core region of each processor (i.e., the hardware exception vectors, the monitor variables and stack, etc.);
- performing busy-wait I/O for the console device;
- managing the processor's watchdog refresh and Multibus timers;
- allowing interactive examination and modification of the contents of the memory, MMU, and the processor's registers;
- providing interactive breakpoint and tracing functions;
- and providing reliable up- and down-loading capabilities via serial lines or by means of the Arpanet TFTP protocol over the Ethernet.

5.1.5. Hardware

The hardware base consists of a collection of processing nodes, connected by a global bus (specifically, an Ethernet). Each node consists of an application processing element and a number of support processing elements. In the current testbed, the application and peripheral processors are Sun Microsystems single board computers. These processing elements feature the Motorola MC68010 microprocessor, one or more megabytes of local memory, a custom MMU (that provides simple segmentation with paging), and number of on-board devices (including programmable counter/timers, a dual USART, and a parallel input port).

It is possible to map regions of the Multibus address space (both memory and I/O) into the processor's virtual address space, and the processor provides autovectored interrupts from on-board devices and from the Multibus. In each node, multiple processors co-exist in a common Multibus backplane and interact via regions of shared Multibus memory and Multibus interrupts. In a minimal configuration, a testbed node is provided with an application processing element, a scheduling processing element, a communication processing element, an Ethernet controller, and 512KB of shared Multibus memory. Chapter 6 provides further details on the testbed hardware.

5.2. Internal Data Structures

A large portion of the kernel's data space, as well as a significant portion of the kernel's functionality, is dedicated to the management of a set of internal data structures. These data structures are used in the management of the system's physical resources and in support of the kernel's programming abstractions. These data structures represent the essence of the resources that the kernel manages, and form the nucleus around which the Alpha kernel is implemented.

5.2.1. Alphabits

The kernel provides its client with an object model to enhance the organization and modularity of applications, and the use of the operation invocation facility provides physical location independent access to the objects in the system. The benefits that the client derives from these features of the kernel are equally useful within the kernel. Frequently, it is the case within the kernel that references are made to entities that may be maintained at remote nodes (e.g., secondary storage images). The support for distribution provided by the operation invocation facility of the kernel is made directly available to the kernel itself by modeling as objects all entities for which remote access is required or may be desirable.

To this end, the kernel maintains data structures, known as Alphabits[1], for each such internal object in the system. An Alphabit is a kernel data structure that exists at the node where the entity it represents resides. Each kernel-level entity (e.g., a thread or an object) has a corresponding Alphabit maintained for it by the kernel. Any entity represented by an Alphabit may be accessed uniformly from any node via the kernel's operation invocation facility. This is true whether the entity is stored locally or remotely, and whether the reference originated from within the kernel or a client object.

---

[1]This feeble attempt at humor is tolerable only when considered in light of the other names proposed for these data structures (e.g., Alpha Particles).

The Alphabit data structures consist of a header, that is common to all Alphabits, followed by fields that are specific to each given type of data structures. The Alphabit's common header contains:

- a globally unique identifier used to access this structure,
- an indication of the type of entity being represented,
- a (logical) timestamp that indicates the last (logical) time at which this structure was last accessed (to be used by the swapping function in selecting entities to be deactivated),
- and a pointer to another Alphabit (to be used for chaining them together in a common hash table entry).

5.2.1.1. Control Blocks

The following is a description of each of the Alphabits currently defined in the Alpha kernel.

System Control Block (SCB) :
This data structure represents the portions of the kernel that use the kernel's virtual memory facility. This Alphabit is used in order to make the kernel appear similar to client objects with respect to its interactions with virtual memory. Those portions of the kernel required to support virtual memory are permanently kept in primary memory (i.e., the pages are "wired"). The pageable portion of the kernel consists of two memory extents — one for the kernel's heap and another for the kernel's virtual page heap. These regions constitute the great majority of the kernel's memory requirements, making most of the kernel pageable. More detail on the virtual memory structure of the kernel is provided in Subsection 5.2.2.

Client Object Control Block (COCB) :
This data structure maintains all of the control and status information for a client object. This includes the object's virtual memory information, the identifiers of those threads that are currently active within the object, the capabilities currently in the possession of the object, and the synchronization information for the semaphores and locks associated with the object.

Client Thread Control Block (CTCB) :
This data structure represents a client thread and contains its control and status information. This includes the virtual memory information for the thread, a pointer to the thread's client stack, a pointer to the Alphabit of the object in which this thread is currently active, and a pointer to the Alphabit of the kernel thread associated with this client thread.

Kernel Object Control Block (KOCB):

This data structure maintains all of the control and status information for a kernel object. This includes a pointer to the kernel object's location in the kernel's virtual address space, the current list of the capabilities associated with the kernel object, and the data structures needed for the object's associated semaphores and locks.

Kernel Thread Control Block (KTCB) :

This data structure provides the internal representation of a kernel thread. This data structure consists of the thread's identifier, a pointer to the thread's kernel stack, the thread's environment, and optional pointers to either a client thread Alphabit or a kernel object Alphabit.

In the current implementation, the thread's environment consists of: the current invocation depth of the thread, the thread's current atomic transaction depth, the current state of the thread (e.g., BLOCKED, STOPPED, or RUNNING), the thread's current state of preemption deferment, the current number of invocations to roll-back on abort, the thread's global importance, the thread's current deadline interval, the kind of deadline that is currently active, the expected amount of computation time required for the current interval, the amount of time that has elapsed since the start of the current interval, and the amount of processing time accumulated by the computation in its current deadline interval.

Storage Object Control Block (SOCB) :

This data structure maintains all of the control and status information for secondary storage entities (i.e., *instance* and *type* objects). This type of Alphabit supports an operation invocation interface between the virtual memory facility and the secondary storage system. These Alphabits provide the information necessary to associate an entity that resides in primary memory (e.g., an object or thread), with a secondary storage entity (i.e., a type or instance object).

This type of Alphabit contains an indication of the kind of entity being represented (e.g., type or instance), the attributes of this entity (e.g., permanent, or atomically-updated), the current state of the entity, and the device-specific addressing information needed to access the entity represented by this Alphabit within secondary storage.

Only those nodes with secondary storage devices will have this type of Alphabit. However, secondary storage objects appear to reside in primary memory, because operations can be invoked on them like any other Alphabit.

As a part of the function of the virtual memory facility, pages that exist in primary memory may be moved to their instance objects in secondary storage (and returned on demand). Furthermore, when objects or threads have not been accessed for some period of time, they may be swapped out to their instance objects, leaving behind only a vestigial Alphabit. This is known as *deactivating* an object and is intended mainly for client objects, which can be swapped out to disk and then reconstructed when referenced again. The deactivation function also permits the migration of objects and threads among nodes in the system (by reactivating them on different nodes).

5.2.1.2. Alphabit Identifiers

Each Alphabit has a globally unique, non-reusable identifier in its header. In the current implementation of Alpha, an Alphabit Identifier is constructed using a simple distributed name generation scheme [Lampson 81]. An Alphabit Identifier is formed by concatenation of the following four fields:

- an indication of the kind of object this is (e.g., individual, inclusively replicated, exclusively replicated object, or system service),
- a unique node identifier (provided by the hardware),
- a count that is stored in permanent storage and incremented each time the node crashes (or when explicitly rolled over),
- and a count that is stored within the kernel's primary memory and incremented each time a new identifier is to be generated.

The size of Alphabit Identifiers was chosen to be large enough to accommodate the number of requests expected during the lifetime of a single execution of the application. An Alphabit Identifier size of 48 bits was chosen because it appears to be large enough to perform realistic experiments with, and yet small enough to be used directly as logical addresses on the Ethernet. Also, the non-volatile portion of the Alphabit Identifier is provided by a battery backed-up time-of-day clock on the application processor board.

5.2.1.3. Dictionary

The Dictionary is a global data structure that contains pointers to all of the Alphabits in the system at any point in time. The Dictionary is implemented as a partitioned, distributed database, where each node has a partition that contains references to its local Alphabits. A local partition of the Dictionary is implemented as a hash table of pointers to local Alphabits, accessed by Alphabit Identifier. As Alphabits are instantiated, pointers to them are entered in the local Dictionaries. Once an Alphabit is looked up in the Dictionary, the Alphabit and data structure it represents can be manipulated.

Figure 5-2 illustrates a local portion of the kernel's Dictionary with one instance of each type of Alphabit entered in it, being indexed by an Alphabit Identifier.

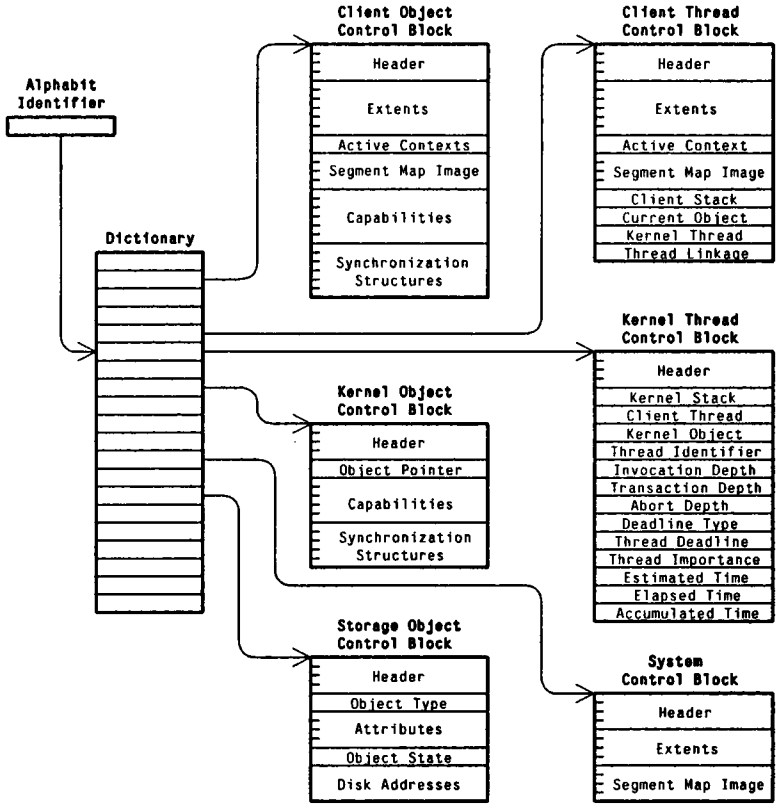

Figure 5-2: Alphabit Control Blocks

5.2.2. Virtual Memory

This Subsection describes the data structures used to support virtual memory in the Alpha kernel. Some of the data structures are provided by the kernel itself, while others are dictated by the underlying hardware. Before these data structures can be described, it is necessary to introduce the hardware structures related to primary memory.

5.2.2.1. Memory Management Hardware

The significant aspects of the underlying hardware are summarized here. All addresses generated by the processor are translated by a custom designed Memory Management Unit (MMU). The structure of the Sun Microsystems version 2.0 MMU is shown in Figure 6-6. At a high level of abstraction, the MMU consists of:

- a Context Register that contains an index into the first of the two translation tables. This index indicates the current address space that the processor is executing in by referring to a contiguous set of memory segments.

- a Segment Map Cache (SMC) that contains a set of descriptors for a set of memory segments. Each segment descriptor contains an index into the next-level translation table, referring to a contiguous set of memory pages.

- a Page Map Cache (PMC) that contains a set of descriptors for a set of memory pages. Each page descriptor contains statistics for the memory page, an indication of the type of page it is, protection information for the page, and an index into the physical memory, pointing to the page of physical memory to which this descriptor refers.

The MMU supports physically separate virtual address spaces known as *contexts*. The Sun 2.0 MMU can store address translation information for a maximum of 8 contexts at any given time. The MC68010 microprocessor has a physical address space of 16MB. Each context is subdivided into 512 segments (corresponding to 32KB's worth of virtual address space), each of which is further subdivided into 16 pages (each of which correspond to 2KB of virtual address space).

This MMU design allows quick context swaps among currently loaded contexts to be performed by simply changing the contents of the context register. The lookup tables used to implement the MMU's address translation are fast, but since they are limited in size, they must be managed like a cache by software. This management is one of the more difficult aspects of the kernel's virtual memory management function.

5.2.2.2. Virtual Memory Requirements

In Alpha, virtual memory is subdivided into a hierarchy of structures starting, at the highest level, with contexts. Each client thread in Alpha is associated with a separate context, and the Alpha kernel itself coexists in the each context along with a client thread. Client objects are not bound to a specific context but rather float among thread contexts. Also, as a result of the MMU caching activity, client threads are bound to different hardware contexts in the course of the system's execution. While the kernel itself shares each context with client threads, the kernel region is protected (for *supervisor* mode access only) so as to disallow access from the client objects (that execute in *user* mode).

Each context in Alpha is partitioned into a set of *memory regions*, that are themselves composed of one or more contiguous areas of virtual memory known as Extents. Each Extent is, in turn, composed of one or more of the virtual memory segments defined by the hardware. Each Extent in the system has associated with it a secondary storage object, that is used to store an image of the Extent for paging or swapping. An Extent may exist completely in primary memory, or may be (entirely or in part) in its related secondary storage object. Figure 5-3 illustrates the decomposition of virtual memory spaces in Alpha.

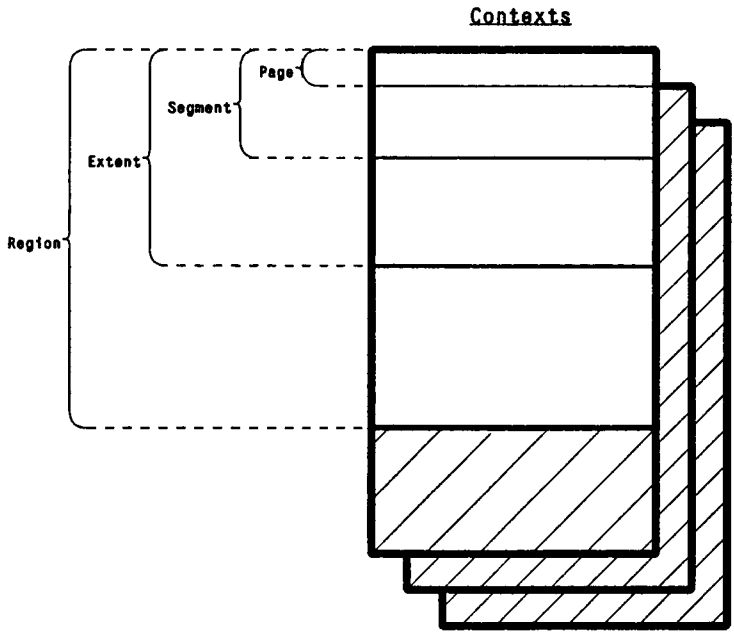

Figure 5-3: Virtual Memory Hierarchy

The virtual memory facility is designed to exploit the system's limited physical resources as efficiently as possible. In the case of the Sun Microsystems version 2.0 processor, the limited resources are physical pages, entries in the SMC, and entries in the PMC. With 1MB of physical memory in the application processing element, there are only 512 physical pages available. The segment descriptors for only 8 contexts can be contained in the SMC at one time, and only 256 sets of page descriptors can be in the PMC at any one time.

In addition to the effective management of these critical resources, the virtual memory facility must

provide the control information necessary for performing each of the kernel's functions, including the invocation of operations on objects, the paging and swapping of objects and threads, and the dynamic creation or deletion of objects and threads. The virtual memory structures must be designed to make it possible to share portions of secondary storage objects among Extents, and for multiple threads to access the Extents that make up an object simultaneously. In order to share them, it must be possible to reclaim the critical memory resources preemptively. It must also be possible to share Extents, both to maintain the consistency of objects that have multiple threads active in them simultaneously, and to utilize the PMC entries efficiently.

Ideally, there should be only one virtual memory structure on each node for each shared Extent, and they should share read-only secondary storage objects wherever possible — e.g., the code Extents of instances of the same type of object should make use of a common secondary storage object. Separate secondary storage objects are required for Extents that contain instance-specific data, and other Extents (e.g., initialized data Extents), may use a common secondary storage objects for their initial page-in requests, and their own individual secondary storage objects for paging out all dirty pages and handling any subsequent page-in requests for modified pages.

The MMU is implemented as a two-level lookup table that is managed by Alpha as a cache for the complete collection of address translation information stored in primary memory. All of the threads and objects in Alpha can be bound to and unbound from different contexts as required in the course of their execution. Threads and objects can dynamically come into, and go out of, existence on a given node, and the virtual memory structures must be able to cope with all of this. The virtual memory structures must also cope with the fact that not every node in Alpha will have secondary storage. The virtual memory structures must also permit the replication of an Extent's secondary storage objects at several different secondary storage sites for reliability and performance reasons.

5.2.2.3. Data Structures

In Alpha, a context contains the kernel, a client thread and a client object. These entities are represented in the kernel by an Alphabit — i.e., a system control block, a client thread control block, and a client object control block. Each of these Alphabits contains a field known as the *segment map image*, that contains the segment descriptors for the Alphabit's portion of a context. Each segment descriptor contains the current state of the segment (e.g., *invalid* or *loaded in this context*), and the PMC index of the segment's page descriptors (if it is currently loaded). When necessary, the SMC is loaded from the segment map image fields of the appropriate entities. Also, when changes are made to segment descriptors in the SMC, they are also made in the segment map image fields in the affected Alphabits.

In addition to the segment map images, these Alphabits contain a collection of pointers to the data structures that represent the Extents which make up the object. This type of data structure is known as an Extent Descriptor, which consists of:

- a pointer to the Alphabit to which this Extent belongs (this is part of the pointer chain needed to permit the page fault handler to identify the owner of a given page)
- the identifier of the secondary storage Alphabit associated with this Extent,
- the starting and ending virtual addresses of this Extent, relative to its context,
- a pointer to the next Extent belonging to the given Alphabit (i.e., the next contiguous Extent in ascending order of virtual addresses),
- and a pointer to a list of data structures that represent each of the segments that make up the Extent.

Sets of linkage pointers assist in the sharing of Extents by objects, and for dealing with the multiple mappings of Extents that occur as the result of threads executing concurrently within a single object. Also, all Alphabits referring to a particular Extent are chained together, and a reference count is maintained to determine the number of Alphabits that are currently making use of the Extent.

The Extent descriptor also serves to link together the virtual memory data structures of an Alphabit (i.e., the Alphabit itself, its Extent Descriptors, and their segment-level data structures) and the secondary storage images associated with its Extents.

The data structure that represents individual segments of virtual memory is known as a PMAP, and contains: a pointer to the Extent Descriptor of which this segment is a part, an array that contains the page map descriptors for this segment, another array that contains the state of each page (e.g., RESIDENT, PAGED_OUT, or IN_TRANSIT), and a pointer to the next PMAP in the Extent.

Just as the segment map image fields of the Alphabits contain the segment descriptors for the SMC, the page map image fields of the PMAP's contain the page descriptors for the PMC. Similarly, the page map descriptors contained in the PMAP must be loaded into the PMC for any of the pages in that segment to be accessed.

Along with the previously described hardware virtual memory structures, the Alpha kernel provides a parallel set of data structures. These kernel-provided *auxiliary* data structures are used to provide additional information for the kernel's virtual memory facility. A decision was made here to trade space for speed, and so additional data structures are provided beyond those logically required, in

order to expedite the virtual memory functions of the kernel. The physical structures that have these auxiliary structures associated with them are physical memory pages, the SMC, and the PMC. The segment map table is a data structure that parallels the SMC, having an entry for each context's set of segment descriptors. Each segment map table entry contains pointers to the client thread Alphabit and the client object Alphabit currently bound to the context. These pointers provide linkages between a context slot in the SMC and the thread and object that are currently loaded in the slot. In addition, each segment map table entry contains an index to another context slot that is used to link the slots together into free and used context lists. The information maintained in the segment map table entries is used in multiplexing the SMC among all of the client threads that are active at a node.

Similarly, a data structure known as the page map table is provided to augment the PMC. Each page map table entry corresponds to a segment's worth of page descriptors in the PMC, and contains a pointer to the PMAP for the segment currently loaded. Each page map table entry also contains an index to permit the free and in-use entries to be linked together by the kernel's virtual memory facility.

A data structure known as the *page state table* parallels the physical memory pages. For each of the application processor's physical memory pages, there exists an entry in the page state table that contains a pointer to the PMAP to which the physical page belongs, and the page's index within the PMAP's page map image. When a physical page is to be manipulated (e.g., for page-out, page-in operations, or on page faults), the page state table provides the information necessary to access the page's virtual memory structures. Additionally, each page state table entry contains an index that is used to link physical pages together for such purposes as maintaining the free page list or the list of pages to be written out to secondary storage.

Figure 5-4 provides an illustration of each of the virtual memory data structures used in Alpha.

5.3. Details of Major Facilities

This Section describes details of the implementation of the major facilities provided by the Alpha kernel. This includes inter-node communication management, virtual memory management, application processor management, and secondary storage management. Because the kernel is replicated at each node, this discussion centers on the functions of a single node.

Machine dependencies exist in most operating system kernels, and the Alpha kernel is no exception. Thus, the following description of the kernel is given in the context of the hardware described in

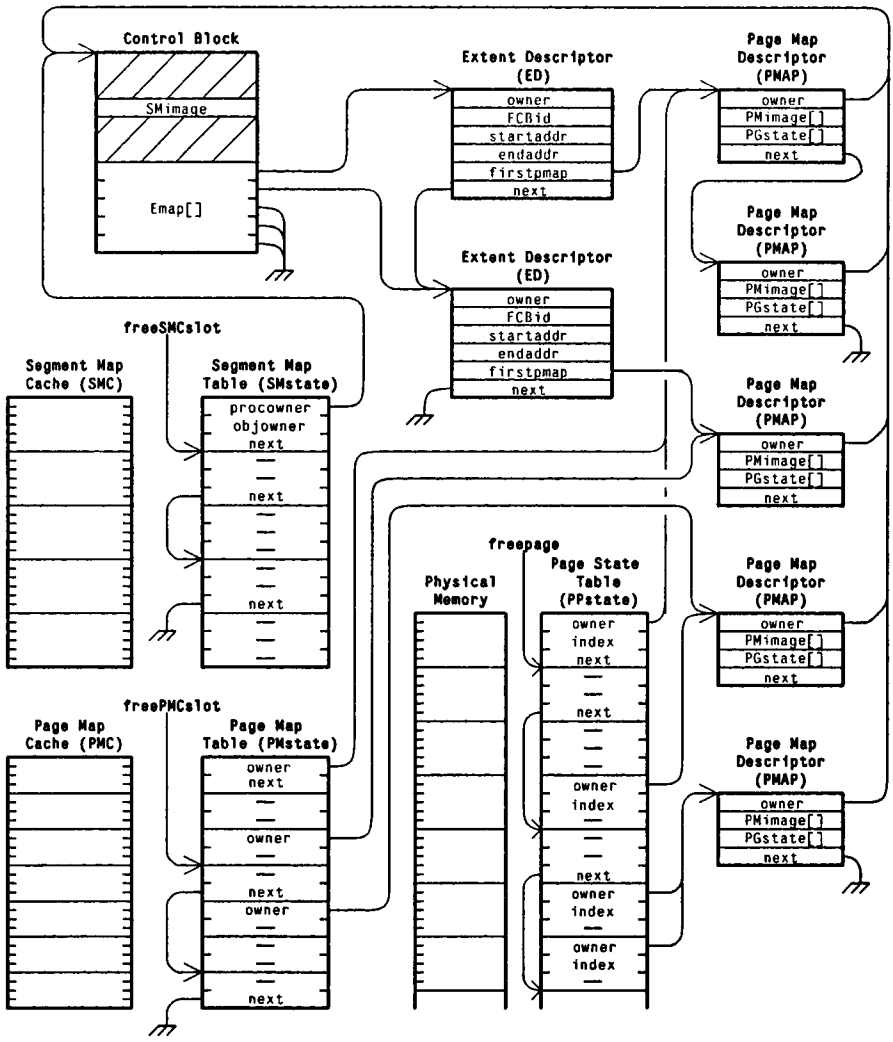

Figure 5-4: Virtual Memory Data Structures

Chapter 6. Although machine dependencies can be found throughout the kernel, the techniques used for its construction are relatively machine-independent. Furthermore, those machine dependencies that do exist within the Alpha kernel are clearly marked and isolated from the rest of the code. As a result, Alpha could be modified with relative ease to work with other loosely-coupled architectures, having processes supporting either a linear, or a segmented, virtual address space and demand-paged virtual memory.

5.3.1. Inter-Node Communication Management

While the communication subsystem may be viewed conceptually as a simple device controller, the hardware that comprises this subsystem includes a separate processing element that executes concurrently with the application processing element. This hardware concurrency allows the communication subsystem to provide the kernel with a highly flexible, high-level interface to the underlying communications subnetwork.

5.3.1.1. Communication Subsystem Interface

The communication subsystem accepts high-level commands from the application processing element, and performs them in parallel with the continued execution of the application processor. The communication subsystem handles per-packet interrupts and interrupts the application processing element only to indicate the occurrence of high-level communications events (e.g., an incoming invocation has arrived or an outgoing invocation has completed). The functions provided by the communication subsystem include message disassembly/reassembly, the intra- and inter-node health maintenance protocols, the atomic transaction refresh protocols, the atomic transaction commit and abort protocols, the system's object replication protocols, and all of the low-level acknowledgement messages.

The communication and application processing elements interact by means of shared memory and interprocessor interrupts. This interface is similar to the hardware-implemented *mailbox* schemes used for interprocessor communication in various systems. Such an interface allows high-bandwidth communication through shared memory, along with the low overhead and high responsiveness of interrupt-based signaling. This interface also provides for increased communication performance, deriving from the ability of the application and communication processing elements to move memory through the manipulation of memory mapping tables, as opposed to performing the costly copying of memory blocks that is normally associated with interprocessor communications activities.

To permit a form of logical addressing to be used, the communication processing element maintains a list of the logical names it recognizes. This structure is examined each time a logically addressed packet arrives, to determine whether the node should accept the packet. Currently this structure is implemented as a hash table. Much of its contents is derived from the kernel's Dictionary and represents the objects and threads that currently exist at the node, while the remainder of the name table consists of the names of logical entities local to the node (e.g., logical clocks and transactions). When the kernel adds or removes entries to or from the Dictionary, the communications subsystem is also notified (via the standard mailbox interface), in order to keep the logical name table up-to-date.

5.3.1.2. Communications Virtual Machine

In order to provide a high degree of flexibility with a minimum amount of overhead, the communication subsystem is designed to provide a virtual machine facility in support of a collection of concurrently executing State Machines (SMs) that perform individual communication activities. All communication protocols implemented by the communication subsystem are implemented as communicating SMs [Bochmann 76, Danthine 80]. The kernel's communication protocols are defined by *protocol templates* that provide the static definitions of communication activities, from which specific instances of SMs are instantiated to handle individual communication activities. Each protocol in Alpha is specified in terms of a state machine description language, which is compiled by a translator program into the SM state tables that constitute the protocol templates. The protocol templates generated by the protocol compiler are linked with the code that implements the communications virtual machine.

The communications virtual machine transforms all communication-related events into *tokens*, which are provided as inputs to the appropriate SM's instances in the virtual machine. Each token serves as an input to one or more SM instances, and each input to an SM instance results in a state transition, with zero or more actions being taken for each transition. Communication events that cause tokens to be generated include interrupts from the application processing element, interrupts from the communication processor's timer, and interrupts from the network controller indicating the receipt or the transmission of a packet. An action may consist of the assignment of a variable (local to the SM instance), or the invocation of one of the mechanisms provided by the communications virtual machine.

The virtual machine includes mechanisms to: copy a block of memory, generate a packet (from a given packet template), queue a packet to be transmitted, deallocate a packet (i.e., return its buffer space to the free pool), deallocate a command block, generate an application processor command block (from a given template), put a command block into the application processing element's mailbox, signal the application processor with an interrupt, initiate or cancel a timed interval event, add or remove names from the communication processing element's logical name table, create an instance of an SM (from a protocol template), and generate a token (from a template).

Additionally, the communications virtual machine is responsible for multiplexing the currently existing SM instances onto the communications processor in order to provide the view that all SMs execute concurrently. This is currently done in a simple time-sliced, round-robin fashion. Future plans include examining the benefits of providing hardware support for the communications virtual

machine to increasing the amount of actual concurrency. Also, an attempt will be made to have the communication subsystem assist in meeting the system's timeliness objectives, by resolving contention for communications resources on the basis of the time constraints of the threads that are associated with the contending communication activities.

This provides for an efficient implementation of the communication subsystem in Alpha. Very little computation is performed at interrupt level; interrupt handlers only generate a token and add it to a SM's input queue. Little overhead is associated with the execution of the individual states of the SMs and the multiplexed execution of the currently active SM instances. The communication protocol SMs are highly responsive, because the functionality of protocols is decomposed into small, uninterruptible, individually schedulable units (similar to Pluribus *strips* [Katsuki 78]). This implementation permits the use of thread attributes to resolve contention effectively, and lends itself well to utilization and scheduling analysis. In addition, this implementation can be provided with hardware support to increase performance through the use of true concurrency.

To simplify the construction of protocols, the communications virtual machine supports nondeterministic SMs. If the inputs to an instance of a SM result in more than one state transition being made, the SM is replicated — i.e., new *versions* of the SM instance are created. All versions of an SM execute in parallel, until a terminal state is reached.

Figure 5-5 provides an illustration of the communications virtual machine, showing all of the virtual machine's facilities and representations of protocol types, instances, and versions.

5.3.1.3. Communication Protocol Specification

On top of the communication virtual machine, the functionality of the communication subsystem is implemented as a hierarchically structured collection of SM instances. The initial demultiplexing of events is handled by the virtual machine; there is an SM defined for each type of input token that may be generated. The lowest-level SM instances serve as handlers for each type of event token that may occur. These SM instances serve to demultiplex the incoming tokens, passing them off to the specific SM instance(s) for which that token is intended. Each entry in a communication subsystem's logical name table has an SM instance associated with it. When a packet is accepted by the communication subsystem, a token is generated (that includes a pointer to the packet) and passed to the SM associated with the logical name it matched. This first-level SM then determines if the token can be handled by it alone, if it should be passed to an existing SM instance, or if a new instance of an SM should be created and the token passed to it. With some protocols, SMs may be involved in

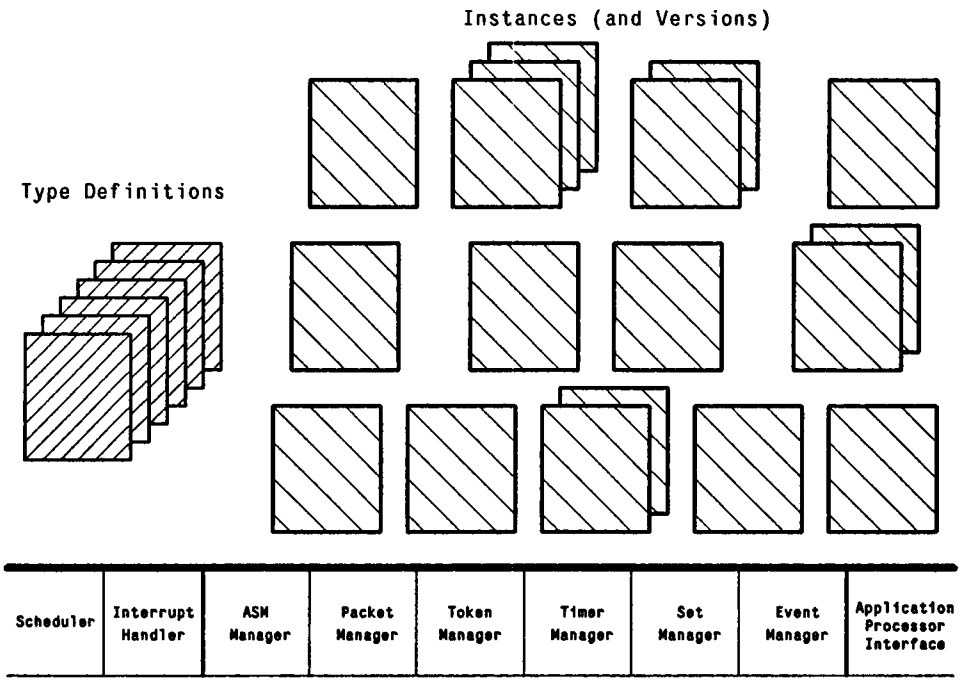

Figure 5-5: The Communications Virtual Machine

multiple conversations (e.g., with eavesdropping or multi-party protocols [Cheriton 84b]). In such cases, a token generated as a result of a received packet may be passed to more than one SM.

When an instance of an SM is instantiated, the communication virtual machine allocates memory for the SM's control block and instance data, and links in a reference to the designated protocol template. When an SM makes a transition into a designated terminal state, it executes the actions for that state and then deallocates itself.

This design of the communication subsystem makes it quite natural to use state-exchange protocols [Fletcher 79] to simplify protocol design. With state-exchange protocols, each participant in an act of communication is modeled as a state machine, and each participant attempts to construct a model that reflects the state of the other parties in the conversation. Each packet that is exchanged contains an encoded indication of the current state of the packet's source, and its view of what state the other parties are in. This explicit exchange of information makes protocol exception handling easier by disambiguating many cases that appear similar in normal protocols.

For example, one protocol frequently used to increase the reliability of the underlying communications subnetwork uses explicit positive acknowledgments with timeouts and packet retransmissions [Fletcher 78]. However, many such protocols result in the duplication of messages at higher levels, because the loss of an acknowledgment is not distinguished from the loss of the original packet. If a simplistic protocol is used, the packet is retransmitted on timeout at the source, which, in the case of the lost acknowledgment, results in a duplicate packet being received. This problem is frequently dealt with through the use of sequence numbers, or some similar scheme, which provides the destination with a means of detecting duplicate packets. With a state-exchange protocol, however, such fault masking does not exist because they are disambiguated by the explicit exchange of state information — i.e., all retries are explicitly marked as such and all acknowledgements refer to a specific instance of a packet. Instead of blindly retransmitting, the communication subsystems exchange explicit information about their states in all of their packets, allowing retries to be detected and simplifying the process of recovering from lost messages.

Figure 5-6 provides an example of a simple protocol defined as a SM. This is a definition of a SM that carries out the various activities associated with the ARPA Internet Address Resolution Protocol [Plummer 82]. This SM is designed to manage a collection of Ethernet/IP address pairs. The SM takes, as inputs, tokens that are generated as a result of packet arrivals, timer interrupts, and commands from the application processing element. Instances of this type of SM handle requests for the Ethernet address associated with a given IP address, or commands to associate this node's Ethernet address with a given IP address. The example SM generates packets to obtain any Ethernet/IP address mappings that it does not have, and it responds to packets that contain requests for Ethernet/IP mappings that the SM has. An additional function provided by this SM is the *aging* of Ethernet/IP address mappings that the SM maintains. When a mapping is added to the SM's list, a timed event is initiated and, when the given amount of time has elapsed, the mapping is deleted.

In this example, the simple protocol is correspondingly small in size, simple to create and easy to understand; the bulk of the example SM description's text consists of protocol description language keywords (shown in bold face). The protocols specified in this way are very stylized, and all of the protocols implemented to date have been easily implemented with the given mechanisms and require very little "random" C code. The SM descriptions for the supported protocols are compiled into the necessary data structures (with a simple translation program), and are linked with the communications virtual machine.

```
/*
 * Example Protocol Definition: ARPA Internet Address Resolution Protocol
 */

/* Constant Definitions */
CONSTANT BC_ENA          0xFFFFFFFFFFFF
CONSTANT MY_ENA          0x012158FFFFFE
CONSTANT MY_IPA          0x7D00FFFE
CONSTANT ARP_REQUEST     0x0001
CONSTANT ARP_RESPONSE    0x0002

/* Data Type Definitions */
DATA_TYPE    ENaddr[8];                    /* Ethernet Address */
DATA_TYPE    IPaddr[6];                    /* Internet Protocol Address */
DATA_TYPE IP_ENelm {                       /* EN/IP Address Pair */
      ENaddr ENA;
      IPaddr IPA;
};

/* Token Type Definitions */
TOKEN_TYPE SETADDR       IP_ENelm;
TOKEN_TYPE GETADDR       IPaddr;
TOKEN_TYPE RCVPKT        ARP_PKT;
TOKEN_TYPE TIMEOUT       IPaddr;

/* Packet Format Definitions */
PACKET_TYPE ARP_IN_RQST_PKT {
      u_short      HAS = 0x0001;           /* Hardware Address Space */
      u_short      PAS = 0x0800;           /* Protocol Address Space */
      u_char       HAL = 0x06;             /* Hardware Address Length */
      u_char       PAL = 0x04;             /* Protocol Address Length */
      u_short      opcode = ARP_REQUEST;   /* Opcode */
      ENaddr       SHA;                    /* Sender's Hardware Address */
      IPaddr       SPA;                    /* Sender's Protocol Address */
      ENaddr       THA = BC_ENA;           /* Target's Hardware Address */
      IPaddr       TPA;                    /* Target's Protocol Address */
};

PACKET_TYPE ARP_OUT_RQST_PKT {
      u_short      HAS = 0x0001;
      u_short      PAS = 0x0800;
      u_char       HAL = 0x06;
      u_char       PAL = 0x04;
      u_short      opcode = ARP_REQUEST;
      ENaddr       SHA = MY_ENA;
      IPaddr       SPA = MY_IPA;
      ENaddr       THA = BC_ENA;
      IPaddr       TPA;
};

PACKET_TYPE ARP_RESP_PKT {
      u_short      HAS = 0x0001;
      u_short      PAS = 0x0800;
      u_char       HAL = 0x06;
      u_char       PAL = 0x04;
      u_short      opcode = ARP_RESPONSE;
      ENaddr       SHA;
      IPaddr       SPA;
      ENaddr       THA = MY_ENA;
      IPaddr       TPA = MY_IPA;
};
```

Figure 5-6: Example Protocol Specification

```
/* Protocol State Machine Description */
SM ARP

STATE Initialize : ACTION { IP_ENset = Create_Set(IP_ENelm); }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE Idle :
       ACTION { NULL; }
       TRANSITION (SETADDR) : (TRUE) => NEXT_STATE SetAddr;
       TRANSITION (GETADDR) : (Member_Set(IP_ENset, TD)) =>
              NEXT_STATE LookupAddr;
       TRANSITION (GETADDR) : (!Member_Set(IP_ENset, TD)) =>
              NEXT_STATE DoARP;
       TRANSITION (RCVPKT) : (Compare_Pkt(ARP_IN_RQST_PKT, TD)) =>
              NEXT_STATE GotRequest;
       TRANSITION (RCVPKT) : (Compare_Pkt(ARP_RESP_PKT, TD)) =>
              NEXT_STATE GotResponse;
       TRANSITION (TIMEOUT) : (TRUE) => NEXT_STATE Timeout;
       TRANSITION (ANY) : (ELSE) => NEXT_STATE Idle;

STATE SetAddr :
       ACTION { Insert_Set(IP_ENset, TD.ENaddr, TD.IPaddr, TD); }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE LookupAddr :
       ACTION {
              IPA = Lookup_Set(IP_ENset, TD);
              Return(IPA);
       }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE DoARP :
       ACTION {
              ARPpkt = Alloc_Pkt(ARP_OUT_RQST_PKT);
              ARPpkt.TPA = TD;
              Send_Pkt(ARPpkt);
       }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE GotRequest :
       ACTION {
              TD.opcode = ARP_RESPONSE;
              TD.THA = TD.SHA;
              TD.TPA = TD.SPA;
              TD.SHA = MY_ENA;
              TD.SPA = MY_IPA;
              Send_Pkt(TD);
       }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE GotResponse :
       ACTION {
              IP_EN.ENA = TD.SHA;
              IP_EN.IPA = TD.SPA;
              Insert_Set(IP_EN);
              Return(TD.SPA);
              Delete_Pkt(TD);
       }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;

STATE Timeout : ACTION { Remove_Set(IP_ENset, TD); }
       TRANSITION (ANY) : (EPSILON) => NEXT_STATE Idle;
```

Figure 5-6, continued

5.3.2. Virtual Memory Management

This describes the manner in which the primary memory of an application processing element is managed by the Alpha kernel. The management strategies applied to physical and virtual memory are defined, as well as the management of the processor's translation tables and the manner in which virtual memory faults are handled.

5.3.2.1. Physical Memory Usage

The Sun Microsystems version 2.0 processor board dedicates separate and complete virtual address spaces to local primary memory, memory mapped on-board devices (including the USART's, timers, the monitor PROM's, and the address translation tables that comprise the MMU), Multibus memory, and Multibus I/O. Each page of a context can map into one of these separate address spaces.

The monitor initializes the processor on power-up, sets the MMU up with an identity mapping of virtual to physical addresses in local memory, invalidating the addresses above the end of the available physical memory, and initializing the necessary memory locations.

The MC68010 microprocessor imposes certain restrictions on the use of the low memory addresses (commonly known as *low-core*). Low-core initialization involves setting up an initial system stack, and installing an initial set of exception vectors (i.e., processor exception vectors, software trap vectors, and interrupt vectors). The monitor also uses portions of low-core to maintain its local variables and a small stack of its own (the *monitor stack*).

Following the power-up initialization performed by the monitor, the kernel is loaded into the system above the defined low-core region (i.e., it is loaded starting at the second segment of all contexts). A page of memory is set aside for the kernel's interrupt stack, then the code and data regions of the kernel are loaded into memory, followed by all of the "wired" kernel threads and objects. These components are combined into a single load-module that is to be loaded into an application processing element when a node powers up (see Figure 5-7). The same load module is used for each node in the system; differentiation based on specific node identifiers occurs following the initialization of the generic kernel.

The kernel's initialization code begins by modifying the MMU tables to construct the desired virtual address mappings for the kernel. The kernel's virtual memory initialization function sets aside the physical pages which contain the low-core information and the load module. These pages are locked in memory (i.e., not available for being paged out) and consist of a contiguous portion of memory at

the bottom of the physical memory space that does not participate in virtual memory paging activities. This approach reduces the initial fragmentation of the physical memory space. The remainder of the physical memory is linked together into the *free page list*, from which pages of physical memory are allocated on demand.

The kernel's pager and swapper daemons (described in Section 5.5.1) are used in an attempt to keep a small number of pages in the free page list at all times. Individual pages of memory can be temporarily exempted from the paging activity. This is known as "sticking" pages down (as opposed to *wiring* them down, which is permanent), and is accomplished by marking the page as such in its PMAP.

When a physical page is to be written out to secondary storage, it is placed into the page-out list to have its contents written out by the pager daemon, and then be returned to the free list. In a similar fashion, a thread's invocation parameter pages are linked together and accessed in a stack order. As the thread makes local invocations, it no longer requires its current incoming invocation parameter page. This page is linked into the thread's list of invocation parameter pages, to be reclaimed when the operation completes and the invocation returns.

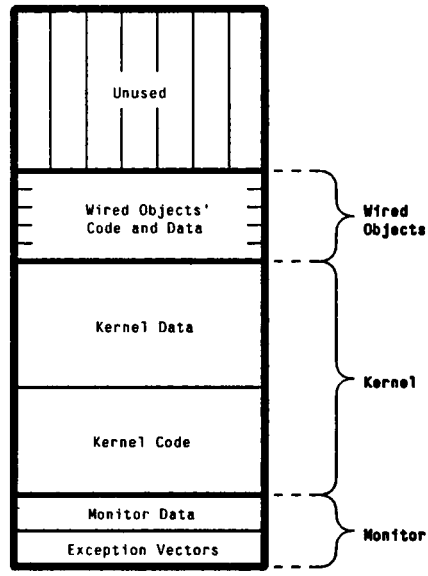

Figure 5-7: Load Module Physical Memory Layout

5.3.2.2. Virtual Memory Usage

Each virtual memory context in Alpha has a common layout. Each context is composed of the following regions: the kernel region, the client thread region, the client object region, and the kernel I/O region. Figure 5-8 illustrates layout of these regions a virtual address space.

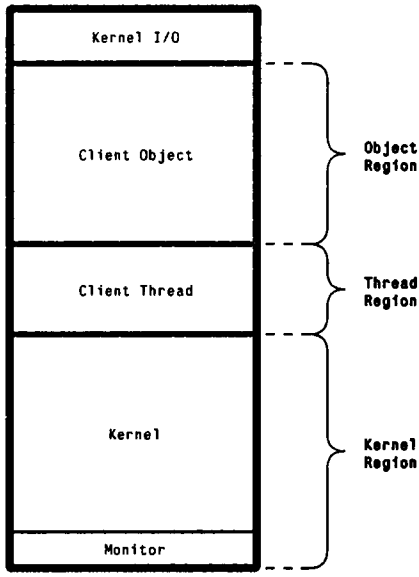

Figure 5-8: Virtual Address Space Layout

— Kernel Region

The kernel region consists of two main parts — the monitor and the kernel proper. The monitor contains the processor's exception vectors, the data needed by the monitor, and a stack used to execute on when executing in extended interrupt handling routines. The monitor Extents are *wired down*, i.e., not available for page replacement and always loaded in the application processor's MMU.

The kernel proper consists of Extents for the kernel code, the kernel data, the kernel heap, and the kernel page heap. The pages that make up each of the different kernel Extents are protected according to their intended purposes — i.e., supervisor mode, execute-only for the kernel code extent and read/write for data, and no access in user mode.

The last two Extents of the kernel are known as the *heap extents*, and are where the kernel (dynamically) maintains most of its major data structures, along with the per-object information for the currently existing threads and objects. The memory contained in each of these Extents is managed according to a heap discipline. The kernel heap Extent provides for the allocation and deallocation of arbitrarily sized units of physical memory within the kernel. The kernel page heap Extent is used to allocate and to deallocate storage on the basis of individual pages of memory. The pages of virtual memory obtained from the page heap can be mapped to actual pages of physical memory, or they can be virtual memory place-holders, into which real pages may or may not be mapped. The memory from the kernel heap is used primarily for the creation of Alphabits and other kernel data structures. Pages from the page heap are primarily used for the kernel stack and parameter pages used by client threads. The Alphabit that represents the kernel (i.e., the SCB) provides the necessary virtual memory structures to permit the kernel page heap Extent to be paged.

The kernel region is in the bottom portion of each context, and this memory is protected to allow access to the kernel region pages only in supervisor mode. All threads execute in user mode, and the kernel is entered through a processor exception instruction (i.e., a trap, an interrupt, or a bus error). Figure 5-9 illustrates the virtual memory layout of the kernel region.

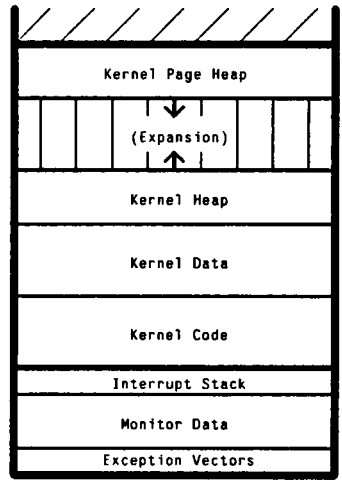

Figure 5-9: Kernel Region Layout

— Client Thread Region

The thread region of a context consists of a single Extent that provides the thread's user mode stack (i.e., the *client stack*) and the client's parameter pages. The thread's client stack is used to provide storage for the automatic variables of threads executing within objects. The client stack is dynamically expandable; it starts out with one page and is expanded each time it overflows, until all pages of the thread's Extent is exhausted. Also, to protect the information contained in client stacks across various operation invocations, new client stack pages are allocated on each invocation and the previous ones are protected to prohibit reading and writing. All client threads execute in user mode, therefore the pages in the client thread Extent are protected in user read/write mode.

In addition to the client stack, the client thread region contains a number of other pages, including a pair of pages used for the invocation parameter pages, and a *guard page* for the client stack. There is one parameter page for outgoing parameters and one for incoming parameters. When a thread is initialized, there is no highest-level invocation, so the incoming parameter page is provided for each new thread by its creating object. The guard page is a single page, placed between the base of the client stack and the parameter pages, that is used to detect the underflow of the client stack. This is done by protecting the page so that any access to it causes a fault. Figure 5-10 illustrates the layout of the client thread region of a context.

For a client thread to execute, it must be loaded into a context within the MMU. To do this, the kernel must access the thread's Alphabit and obtain the necessary segment descriptors to load into the SMC. The act of acquiring a context for a thread to execute in may involve the unbinding of another thread from its context to make room in the SMC for the incoming process. Also, just as pages of physical memory can be wired down, so too can certain contexts be made ineligible for removal from the SMC.

— Client Object Region

The client object region is where a thread's currently active object is placed. When an operation is invoked (on a local object) by a thread, the kernel is entered, the client object currently mapped into the thread's context is mapped out, then the destination object is located and mapped into the invoking thread's context. Then, the appropriate entry point into the object is determined and the thread resumes execution within the new object. This process is reversed when an operation completes. While parameter pages are kept in the thread's client stack Extent, they are accessible to the client object, and are used to pass parameters on operation invocation.

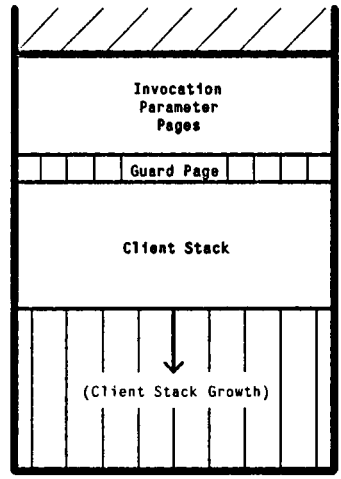

Figure 5-10: Client Thread Layout

The client object region consists of three Extents — the object code Extent, object data Extent, and the object heap Extent. The client object's code Extent contains the object's code and is protected in a user execute-only mode. The client object data Extent contains the object's data (both initialized and uninitialized), and is user read/write protected. The object heap Extent is where dynamic storage is allocated for the object and is also protected in user read/write mode. Figure 5-11 provides an illustration of a client thread's region of a context.

Like client threads, a given client object may at times not be bound to any context. Unlike client threads however, client objects may exist in multiple contexts at any one point in time. Therefore, the client object Alphabits must contain references to all of the contexts that the object is currently bound to, as opposed to the reference to the single context required by client thread Alphabits.

— Kernel I/O Region

The final memory region in a context is known as the kernel I/O region. This region is logically a part of the kernel region, but because of hardware constraints, is in a separate location (i.e., exception vectors must be in low-core, and DVMA space is at the top of the virtual address space). This region is managed by the kernel, is not paged, and is protected in supervisor read/write mode. The processor's on-board I/O devices, PROM's, and DVMA accessible memory is mapped into this region.

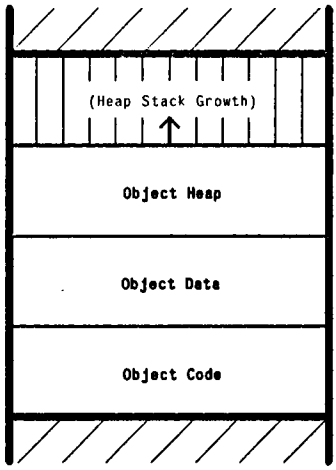

Figure 5-11: Client Object Layout

5.3.2.3. Address Translation Table Usage

The MMU's SMC and PMC are not large enough to contain all of the segment map and page map entries required for every context that must be supported by the kernel. As a result, the actual virtual memory addressing information is stored in the kernel's virtual memory data structures, and the translation tables are multiplexed among the entities active at a particular node.

In Alpha, the MMU is managed like a cache, and the complete virtual memory information is kept in other data structures within primary memory. The actual virtual memory data structures are always kept up-to-date, and thus it is not necessary to write out MMU descriptors before they are reused. The union of all of the segment map image fields in Alphabits, and page map image fields in PMAP's constitute the system's virtual memory data structures. This is a distributed version of the type of memory translation data structure that is usually found in systems where the MMU is implemented as a hardware cache. Because there is no hardware support for the translation cache, there is little point in providing a centralized data structure for the translation information. This distributed approach provides a simple means of managing the necessary mapping information, and simplifies some of the problems associated with the sharing of code.

All modifications of the SMC and PMC are handled by the kernel. The possible reasons for manipulating the MMU include: segment and page faults, the invocation of operations, and binding

a thread to a context. When an object or thread is to be loaded into a context, its segment map image is taken from its control block and loaded into the SMC. The necessary page map information is not loaded when a thread is bound to a context, but rather is faulted into the PMC on demand.

In order to set up a client thread's context, the kernel must first enter the new context. This is because hardware restrictions allow modification of the translation table mappings only for the current context. Thus a performance penalty is incurred each time the virtual memory mappings of a client context are modified by the kernel.

The MMU caches the address translation information contained in Alphabits, and the MMU's tables must be multiplexed among all of the threads active at a given node. To make effective use of the critical resources in the MMU, it must be possible to preemptively remove the segment and page descriptors from the MMU. Similarly, it must be possible for the virtual memory facility to restore segment and page descriptors to the MMU on demand (displacing other descriptors where necessary). Whenever translation table entries are removed from the caches in the MMU, they must be saved in their primary memory data structures so that they can be restored at a later time.

When a translation table entry is needed in the MMU, and a table is full, one of the existing entries must be selected for replacement. Usually, manipulation of the MMU tables are done on sets of descriptors (i.e., a context's worth of segment descriptors, or a segment's worth of page descriptors). A simple, global (i.e., across the entire cache for a given node) FIFO replacement scheme is used to select the set of descriptors (or *slot*) that is to be replaced from among the set currently eligible for replacement. After a slot has been selected, its information is copied to the necessary Alphabits so that the new information can be loaded into its place in the cache.

Some MMU table entries are not eligible for replacement because the information they represent must be mapped at all times. Examples of these types of entries are the segments comprising the monitor, kernel code, and kernel data Extents. Other translation table entries must be bound to the MMU for short periods of time, but are otherwise eligible for replacement. Most notable of these are the descriptors for the segments containing the kernel stack for the currently active thread. Those MMU slots that are never eligible for replacement are removed from the eligible collection at initialization, while the kernel provides a data structure with which those slots that are temporarily ineligible for replacement can be *stuck down* by the kernel.

5.3.2.4. Virtual Memory Fault Handling

The kernel is responsible for handling all virtual memory faults. When a fault occurs, a bus error trap occurs that switches the processor to supervisor mode and begins execution within the kernel's virtual memory fault handler. Once in the kernel, a determination is made of whether the fault is a segment or a page fault. In the case of a segment fault, the specific type of fault is identified — e.g., an illegal reference, a request to load a segment map, a signal to extend a client stack segment, to fill a page with zeros, or to provide a physical page for a thread's outgoing parameter block page. An illegal memory reference made by a client object terminates the currently active thread, while an illegal reference generated by the kernel causes the application processor to trap to the monitor. In either case, an illegal (page or segment) reference causes a message to be printed on the node's console.

If the bus error indicates a page fault, the nature of the fault is determined — e.g., an illegal reference, a stack extension request, a demand paging request, or a request to provide a parameter page. Illegal page references are handled in a manner similar to how illegal segment references are handled.

Since PMC slots are reclaimed according to a global FIFO replacement policy, a segment fault can result in one PMC slot being emptied in order to accommodate a faulting reference. Also, since a thread's user and supervisor stack pointers must always be valid, the victim selection algorithm of the PMC manager must never select a segment containing the currently executing thread's kernel stack. To insure that this is the case, the notion of a *sticky* PMC slot is introduced — i.e., a slot that is not eligible for replacement. When a thread executes, the kernel always *sticks down* the current thread's stack segments. These segments are *unstuck* when the thread is descheduled.

In the event of a client stack extension request, the stack is automatically extended (by pages and segments) as long as there is memory space. When there is no more (virtual or physical) memory for the client stack, then the stack extension fault is interpreted as an error condition. Also, when a fault indicates that parameter page is to be supplied by the kernel, a physical page is obtained from the free page list and installed into the appropriate thread's context.

5.3.3. Application Processor Management

Each time a DEADLINE operation is invoked on a thread, the kernel evaluates the deadline parameters, and modifies the thread's environment only if the new deadline is more constraining than the currently active deadline. Likewise, when a thread completes a deadline block (i.e., executes the MARK operation of the thread), the kernel determines whether the thread's environment should be altered to reflect a less constraining outer-level deadline. The deadline information in a thread's environment is saved and restored, in a stack-oriented fashion, in the course of invoking nested deadlines.

The deadlines placed on the execution of portions of threads are specified in terms of some integer number of microseconds. When a DEADLINE operation is invoked, the deadline is converted into an absolute time relative to the local node's real-time clock. This information is sufficient as long as the thread remains on the same node. However, when threads move among nodes (either as a result of a remote operation invocation, or the migration of a thread), their specified deadlines must be valid with respect to the destination node's clock. The maintenance of clock synchronization among all of the nodes in the system was considered an unreasonable attempt at simulating the behavior of a centralized system, thus the deadlines for threads must be adjusted when threads move among nodes. For this reason, the time required to move threads among nodes must be provided to the kernel. Despite the fact that the current communications subnetwork in Alpha (i.e., Ethernet) does not support this functionality, the communications subsystem provides reasonable estimates of thread transfer times for use by the kernel.

The need for thread transfer time information is example of where a communication subnetwork that is oriented towards real-time applications could provide support for a real-time system. Also, this kind of support is more significant to real-time systems than just the deterministic resolution of contention for communications resources sought after in "real-time" communication subnetworks.

The Alpha kernel makes use of measured elapsed times, as opposed to constant time estimates, in many resource management functions (e.g., message transfer times in the communication subsystem or device access times in the secondary storage subsystem). This differs from the approach taken by other real-time systems that attempt to constrain the system's behavior in order to make it conform closely to the designer's static assumptions. Since systems cannot be made error-free, and actual delays cannot be kept constant, more adaptive techniques are (in general) better suited for use in practical systems than schemes based on attempts at making *a priori*, absolute guarantees.

The interface to the scheduling processing element was designed so that the scheduling facility can

perform its function external to the application processing element being managed. The interface between the kernel (which executes on the application processing element) and the scheduling subsystem is composed of a collection of commands that are exchanged via a node's interprocessor communications mechanism. The scheduling processing element has commands that allow it to control the binding and unbinding of threads to and from the application processor. While the application processing element has commands for transferring the information needed for the scheduling function to the scheduling processing element.

The scheduling subsystem issues a command to the application processing element to indicate that the current thread should be preempted. The application processing element responds to a preemption command by suspending the execution of the currently executing thread, and issuing a command to the scheduling subsystem to indicate that the thread has been suspended. When the scheduling subsystem receives a command indicating that the thread has been preempted, it adjusts the accumulated computation time for the suspended thread, and responds to the application processing element with a command that includes a reference to the Alphabit of the next thread to be executed.

In addition, the application processing element has commands that allow it to indicate changes in a thread's state. These commands can be used to indicate that a thread should be added to, or removed from, the Ready Queue. These commands include information about why these manipulations should be performed — e.g., a thread has blocked on a semaphore or a lock, a thread has been suspended because of a virtual memory fault, a thread has been frozen or deleted, or a thread has been blocked while it makes an invocation on a remote object. There is also a command that allows the application processing element to inform the scheduling subsystem of changes that occur in the environment information of a thread currently in the Ready Queue.

The scheduling subsystem interface has also been designed to allow the scheduling facility to execute on the application processors, should the scheduling algorithm being used not need to be evaluated concurrently with application processing (e.g., with simple round-robin or priority scheduling schemes).

Additionally, the scheduling subsystem can use the information it maintains concerning threads to determine which of the waiting threads should be unblocked when a V operation is performed on a semaphore or an UNLOCK operation is performed on a lock. In this way, the thread to unblock can be chosen in a fashion that is based on the same information and policies used in scheduling the application processing element.

5.3.4. Secondary Storage Management

In Alpha, secondary memory is managed by the kernel's secondary storage facility. This facility is internal to the kernel (i.e., does not have an interface that is directly accessible to the clients of the kernel) and provides support for the object abstraction. The major function of the secondary storage facility is to provide and to maintain images of primary memory Extents, within secondary storage.

Each Extent in Alpha has a *storage object* associated with it (pointed to by a field in Extent Descriptors). A storage object acts as a repository for one or more Extents. Each storage object is physically located in either the transient or permanent secondary storage provided by the secondary storage facility, and the images may be updated atomically. These features of secondary storage provide support for the attributes associated with the object programming abstraction.

5.3.4.1. Secondary Storage Facility Interface

The secondary storage facility provides an object-like interface to the rest of the kernel, thereby allowing the operation invocation facility to be used to access the storage objects maintained by the secondary storage facility. The clients of the secondary storage facility therefore need not know the physical location of the storage objects it accesses, they need not be aware of whether a storage object is permanent or atomically updateable, nor must clients know if the storage object is replicated at several secondary storage sites.

Because the kernel uses the operation invocation facility to access storage objects, the secondary storage facility supports the set of operations that may be performed on secondary storage facility and storage objects. The secondary storage facility appears as a kernel object (with a fixed Alphabit Identifier, like those of system service objects), and each storage object is given its own unique Alphabit. The secondary storage facility handles the operation invocations directed towards the facility as well as those addressed to individual storage objects.

The operations defined on the secondary storage facility are:

DEFINE_TYPE: This operation creates a new type storage object, and associates with it the given type identifier. This operation fails if there is insufficient secondary storage to create another storage object, or if there already exists a type storage object with the given name. If successful, this operation creates the Alphabit control block for this storage object, enters it in the local Dictionary, and returns the Alphabit Identifier for the newly created storage object. In the Alpha kernel, it is not anticipated that new types will be created at run-time. This operation is provided primarily for the adding new types to the secondary storage system in an "off-line" fashion.

INSTANTIATE: This operation creates a new instance of a storage object, and associates it with the type storage object that is given as a parameter. This operation returns a failure indication if there is insufficient space to create another storage object, or if there is no type storage object associated with the given identifier. If successful, this operation creates the Alphabit control block for this storage object, enters it in the local Dictionary, and returns the Alphabit Identifier for the newly created storage object.

DELETE: This operation is used to remove a specified storage object (of either the type of instance variety). This operation returns a failure indication if the storage object, specified as a parameter is not found. If successful, this operation deallocates the secondary storage associated with the given storage object and deallocates the Alphabit associated with it in the kernel.

In Alpha, the secondary storage facility is implemented on those nodes with secondary storage subsystems (i.e., the nodes that have secondary storage devices and secondary storage processing elements). The kernel's secondary storage facility behaves as an exclusively replicated client object. Secondary storage operations are performed on a local instance of the secondary storage facility where possible, and from a remotely chosen location otherwise.

Secondary storage devices are connected to nodes in Alpha in much the same way as nodes are connected to the communication subnetwork. Each node's secondary storage subsystem in Alpha consists of a hard disk, a disk controller, and a secondary storage processing element. Taken together, these hardware units can be viewed as a disk with a programmable, caching controller, just as the communication processor and the Ethernet interface in the communication subsystem can be considered an intelligent network interface. Figure 5-12 provides an indication of the relationship of the secondary storage devices to the testbed system.

The secondary storage processing element maintains all of the necessary directory information needed to service the requests made on the facility. The secondary storage subsystem is responsible for managing all of the directory-type information that is necessary to save and retrieve pages to and from storage objects. Also, the secondary storage processor is responsible for performing all of the caching of control and data information associated with storage objects. It is responsible for "short-circuiting" pages that are queued to be paged-out when a page-in request arrives for them. The secondary storage facility is also responsible for recognizing situations in which pages do not have to be written in order to fulfill page write operations. This may occur when the page to be written is a read only page (e.g., a code page from a type object), or if the page has not been modified since the last time it was read from its storage object. Additionally, the concurrency provided by the secondary storage processing element makes possible a number of optimizations. For example, the secondary

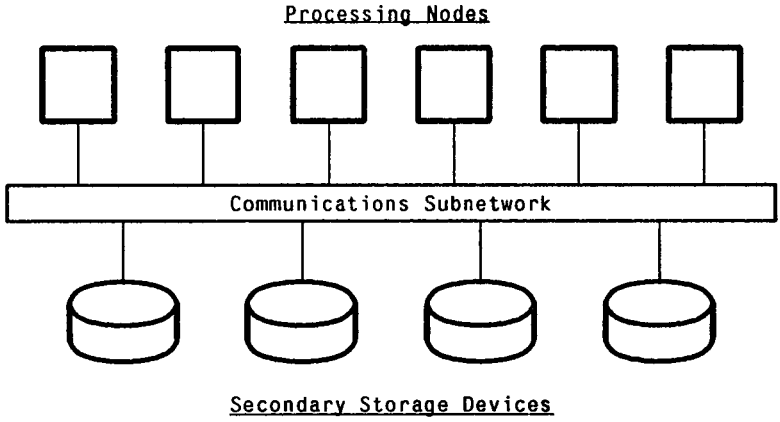

Figure 5-12: Logical View of Secondary Storage in Alpha

storage subsystem could not only maintain information concerning the physical layout of the information on disks, but could also monitor the position of the heads and the disk's rotational position in order to increase the performance of the subsystem.

5.3.4.2. Secondary Storage Objects

There are two varieties of storage objects: *type* and *instance* storage objects. Secondary storage objects are registered in the Dictionary of the node at which they exist, just like the objects in primary memory. When the kernel starts up, the only storage objects that exist are those that possess the permanence attribute — type objects and permanent instance objects. Furthermore, when the kernel starts up, updates are completed on all storage objects with the atomic update property that were in the process of being updated when a node failure occurred.

Instance storage objects get their code and initialized data pages from the specified type object, and any number of instance storage objects can share a common type storage object. When the kernel wishes to move portions of an object's Extent into or out of primary memory, the Alphabit Identifier found in the Extent Descriptor is used in performing an invocation on the appropriate instance storage object.

The operations defined by the secondary storage facility on storage objects are:

PAGE_IN: This operation reads the specified page from a given instance storage object, into the specified physical page, and installs the page in the proper location in virtual memory. This operation fails if the specified storage object cannot be found, or if the specified page is not valid within the given storage object.

PAGE OUT: This operation writes the specified physical page from primary memory into its instance storage object. This operation fails if the specified physical page is not associated with an extent, or if secondary storage is exhausted. If the instance storage object is atomically updatable, the page is not written directly to the storage object, but to a buffer area in the same type of storage (i.e., transient or permanent), and the write is actually completed when an **UPDATE** operation is performed. Additionally, this operation takes a parameter that indicates whether the page to be written out has locks associated with it. If so, this operation uses the data in the log associated with each lock, in place of that which is actually in the physical page at the time.

UPDATE: This operation causes all of the changes made to an object to be performed in such a manner as to not allow the object to be visible in a partially updated state. This operation has no effect if the storage object upon which it is being invoked does not have the atomic update property. This operation deals with all of the pages buffered by the **PAGE_OUT** operations since the last **UPDATE** operation was performed. The effects of this operation are achieved by linking in all of the buffered pages, atomically with respect to read operations invoked on the storage object. If the object on which this operation is being performed has the attribute of permanence, the change is registered in a type of *intentions list* and multiple reads and writes are used to ensure that all of the changes are made, even across node failures.

It is less expensive to manage the transient portion of secondary storage than it is to manage the permanent portion. This is because no redundancy is required for the disk directories of transient storage, the disk sectors of transient storage can be heavily cached, and the cached sectors do not have to be written to disk except to make room in the cache. On failure, the transient portion of an object must be reinitialized, but no complicated recovery procedures are required.

The permanent part of secondary storage is managed in a manner similar to that used in traditional file systems. Redundant disk directory information is maintained on the disk to keep from losing objects as a result of a node failure. Care must be taken in performing reads and writes to ensure that once an object has been written, it will retain its state across failures. To this end, a technique similar to that for providing stable storage as described in [Lampson 81] is used. This makes the read and write operations on permanent storage objects more costly than for transient storage objects.

After a failure, the permanent portion of secondary storage must be restored to a consistent state, Alphabits must be created in primary memory, initialized, and entered in the local node's Dictionary for all of the permanent objects in the recovering node's secondary storage. All pending atomic update operations on permanent objects must be completed before references to them are placed in the Dictionary.

While the attributes associated with objects are usually defined when they are created, it is possible to change object attributes at run time. These changes must be reflected in the instance storage objects. For example, when a transient object is converted into a permanent object, a permanent storage object must be created, the contents of the transient storage object must be copied into it, the object identifier in the proper Extent descriptors must be changed, and the transient object should be deleted.

The operations performed on secondary storage objects involve accessing secondary storage devices, that typically have characteristics that necessitate special treatment (e.g., average access times on the order of 30 milliseconds for disks). To deal with this, special management algorithms are frequently used (e.g., to minimize head motion) which make it necessary to handle secondary storage operations in a sequence different from that in which they were requested. Furthermore, the same thread environment information that is used by the scheduling subsystem to resolve contention for processing cycles is used by the secondary storage subsystem to resolve contention for the secondary storage resources. In particular, the order in which disk access requests are serviced is influenced by the characteristics of the threads making the requests.

When an operation is performed on a storage object, the secondary storage facility ensures that the invocation returns as soon as possible. In some cases, the request embodied by the operation invocation has been completed when the invocation returns, while in other cases the return indicates only that the request has been noted and it has been queued. As an optimization, the reply from invocations of operations on storage objects has been decoupled from the indication of the completion of the operation. When an operation on a storage object actually completes, the secondary storage facility executes a **V** operation on a semaphore that is associated with the thread whose request has been satisfied. For this purpose, each thread Alphabit has associated with it a semaphore dedicated to the invocation of secondary storage operations. Should the thread then wish to wait until the request has completed, it can issue a **P** operation on its secondary storage semaphore. A **C_P** operation can be issued by the thread to poll for the completion of the secondary storage operation.

When a thread page faults, the faulting thread enters the kernel via the fault handler and invokes a **PAGE_IN** operation of its instance object. Upon successful return from the invocation, the thread immediately issues a **P** operation on its secondary storage semaphore. If the page has not yet been installed, the thread is blocked and will be unblocked by the kernel when the page has been installed. In this approach, the thread does not automatically relinquish the processor when it makes a system call that could take a long time. Thus, if the requested page is easily available, the secondary storage

facility installs it and issues a V operation on the thread's semaphore prior to returning from the invocation, thereby not incurring the overhead of rescheduling the thread. On the other hand, the thread is blocked if the secondary storage facility cannot ensure the prompt installation of the requested page.

The page daemon's thread (see Subsection 5.5.1) is typically blocked until the level of the physical page pool crosses its lower limit. When this happens, the page daemon thread is unblocked and goes on to identify a number of victim pages to be written to secondary storage. The page daemon thread invokes a PAGE_OUT operation on the instance object of each victim it chooses. The page daemon need not wait for each request to complete, instead, it issues a C_P operation on all but the last of its page-out requests, issuing a P operation following the last request. In this fashion, the page daemon can queue a large number of page-out requests, have them serviced in arbitrary order, and wait for them all to complete before continuing. Once all of the secondary storage operations have been performed, the page daemon may then become blocked once again until the pool's level crosses its lower threshold.

5.4. Details of Kernel Facilities

This section describes various details concerning the implementation of the major kernel facilities. The specific details related to the following facilities given here are: object management, operation invocation, thread management, access control, and concurrency control.

5.4.1. Object Management

While the clients of Alpha view the kernel object optimization as just another attribute of objects, the implementation of kernel objects is quite different from that of client objects. Both kinds of object type specifications are written the same, the only difference is that the KERNEL qualifier is used in the header to indicate that instances of objects of this type are implemented as kernel objects.

Both client and kernel objects are composed of the same major logical components — i.e., code, data, and a c-list. With client objects the code and data components exist in separate virtual memory extents, that are mapped into and out of thread contexts on invocations. In the case of kernel objects, the code and data of all kernel objects are all combined within the code and data Extents of the kernel. Kernel objects are an integral part of the kernel, and do not have separate virtual memory structures or their own storage objects associated with them. For this reason, kernel object Alphabits do not contain references to Extent Descriptors, SMC information, or the context(s) the object is

currently mapped into. The KOCB is quite simple and consists mainly of a pointer to the object's entry point block in the kernel's data Extent, the kernel object's c-list, and its synchronization data structures.

Just as with client objects, the object language pre-processor generates an entry point block for each kernel object. The object entry point block contains a count of the number of entry points into the object, followed by a list of addresses of the entry points. This is used by the operation invocation facility to validate entry points into objects. However, entry point blocks for client objects are placed at the start of each object's data Extent, while the entry blocks for kernel objects are placed in arbitrary locations in the kernel's data Extent.

In addition to linking kernel objects differently (i.e., with the kernel, as opposed to being a separate entity), the object language pre-processor generates different code for the operation invocations made by kernel objects. Since the kernel objects all execute in the kernel context and in supervisor mode, the usual trap instruction used by client objects to gain access into the kernel on invocations is not used. Instead, the pre-processor generates the code to marshal the necessary parameters and then to make a call to the kernel's operation invocation routine.

5.4.2. Operation Invocation

The operation invocation facility has been implemented to provide a uniform interface to programmers. Operation invocations appear the same regardless of the type of object (i.e., client or kernel) making the invocation and the type of object being invoked — i.e., kernel or client object or thread, system service object, replicated object, local or remote.

The object language pre-processor generates code to handle operation invocations. An invocation begins with the marshaling of parameters — done by packing all of the passed variables into the invoking object's invocation parameter page, followed by the indices for all of the passed capabilities. The object language pre-processor generates code, at both the source and destination of invocations, to pack and unpack the parameters and capability indices. Because the current testbed consists of a collection of homogeneous application processors, there is no need for the transformation of the parameters. However, the object language pre-processor provides hooks to add such translation routines if the need should arise in the future. If a real programming language were provided, the marshaling of parameters could be simplified considerably by allocating the allocating space directly in the parameter pages for the invocation parameters.

When the parameter pages have been loaded with the invocation parameters, the object traps into the kernel. This causes the application processing element to enter supervisor mode and begin the invocation process. The very first thing that is done by the operation invocation trap handler is to determine whether it is a *special* invocation, i.e., one in support of one of the object programming language primitives. This is determined by the value of the destination capability. All objects can invoke these operations and so there is no need to perform the normal capability translation and validation steps. This invocation path represents the first of a series of "short-circuit"-style optimizations that the kernel the provides to compensate for the performance costs of unifying the kernel's interface with operation invocation.

If a standard operation is being invoked, the invocation procedure continues with the translation and validation of any capabilities passed as parameters. This is described in detail in Subsection 5.4.4, but briefly, this involves applying the passed capability indices to the invoking object's c-list to obtain the desired capability *descriptors*, and examining them to determine if there are any restrictions applied to the capabilities that would make their desired use illegal. In each invocation there is at least one capability that must be translated and validated, and that is the capability used to indicate the object that is the destination of the invocation. If the passed capabilities are in any way improper (e.g., non-existent or usage restricted), the invocation is terminated and a failure indication is returned. When all of the capabilities are successfully translated, the invocation proceeds by calling the internal invocation routine.

At this point in the operation invocation process, an internal operation invocation interface is used. This interface is provided by the kernel for use both by objects, and the kernel itself, to perform location-independent invocation of operations on a variety of kernel entities (i.e., anything represented by Alphabits). This routine takes a pointer to an operation invocation parameter page (in a standard format) and continues with the operation invocation, and is used by all operation invocations from client objects, kernel objects, or functions of the kernel itself. The internal operation invocation routine begins by determining whether the destination of the invocation is a local system service object. If so, the system service object handler is called, otherwise a lookup operation is performed on the local Dictionary to determine if the destination Alphabit is local to this node. If the destination entity's Alphabit Identifier is found in the local node's Dictionary, the routine that handles invocations on local entities is called. If the destination entity is not found locally, the routine that interacts with the communication subsystem is called to issue a remote invocation. (An exception to this procedure is made in the case of inclusively replicated entities, where the remote invocation is done regardless of whether the destination entity exists locally.)

The code that handles local invocations determines the type of entity on which the operation is to be performed and calls the appropriate routine for each of the potential destination types. This routine is invoked either as a result of a locally initiated operation invocation, or as a result of an incoming remote operation invocation (in which case it is initiated by a signal from the communication subsystem). Among the routines that handle the local invocations are routines that provide the standard operations on threads and secondary storage objects, in addition to the two main routines that handle invocations on kernel objects and client objects.

The part of the kernel that performs the invocation of operations on kernel objects begins by adding any passed capabilities to the invoked object's c-list, and replacing the capabilities in the invoked object's operation invocation parameter page with their indices relative to the object's c-list. Following this, the entry point of the desired operation is obtained by locating the object's entry point block (as referenced by the object's KOCB), determining if the operation index is within legal bounds, and then indexing into the entry point block to get the address of the operation entry point. Once the invoked operation's entry point has been obtained, the invoked object is entered and execution begun at the start of the desired operation. Note that regardless of the context from which the invocation originated, all operations on kernel objects are performed within the kernel.

The code that is responsible for performing operation invocations on client objects functions similarly to the one that handles kernel object invocations. However, a number of additional activities must be performed in order to initiate an operation on a client object. The kernel routine must first determine whether the operation invocation has a client context associated with it. If not (e.g., if the operation invocation came from a kernel thread in a kernel object), a context must be created before the invocation can be made on the client object; only when the kernel has allocated and initialized a client context, can the invocation precede. When it has ensured that a client context exists, the client object invocation routine locates the virtual memory structures associated with the destination client object, switches to the invoking thread's context, maps the invoking client object out of the invoking thread's context (if there was one), maps the invoked client object into the context, locates and validates the operation's entry point, and begins execution within the invoked object's context. This is done by creating a dummy stack frame, and executing a return from exception instruction to begin execution (in user mode) of the given operation, in the invoked object.

If the destination entity's Alphabit Identifier is not found in the local Dictionary, a routine is called that creates a command block, puts it in the communication subsystem's mailbox, and signals the communication subsystem. The command block contains a opcode indicating that this is to be a

remote invocation, a pointer to the passed the invocation parameter page, the Alphabit Identifiers of the invoking object and thread, and the invoking thread's environment. Once the remote invocation routine passes this request to the communication subsystem, the kernel blocks the invoking thread, and continues with the execution of another thread. When the remote invocation completes (successfully or otherwise), the communication subsystem places the reply information in shared memory, a response command block is placed in the application processing element's mailbox, and the communication subsystem interrupts the application processing element. When the kernel receives such a signal from the communication subsystem, it locates the reply information, makes the necessary changes to the invoking thread's environment, composes an operation invocation response (to appear as though it came from a local object), and unblocks the invoking thread. When the invoking thread runs again, the invocation completes in the normal fashion.

Remote invocations create surrogate kernel threads at the destination node, that are obtained from a pool of preallocated resources (as described in Section 5.5). A surrogate thread inherits the invoking thread's environment and then invokes the operation on the object specified in the incoming parameter page provided by the communication subsystem. From that point on, the incoming remote invocation is handled just as if it had originated from a local kernel object and kernel thread. The surrogate thread is added to the remote node's scheduling mix and is scheduled based on its inherited environment information. Once a remote operation invocation completes, the communication subsystem is given the results, along with any changes to the thread's environment. This information is passed back to the invoking object, and the surrogate thread is deactivated.

In all of the cases outlined above, once an invoked operation completes, the sequence of steps taken is retraced to return to the invoking object, passing back any results from the invocation. The return parameters are passed back in the reply portion of the parameter block page, and all manipulations of virtual memory are reversed.

Figure 5-13 illustrates the routines that make up the operation invocation facility and their relationships with each other.

5.4.3. Thread Management

A kernel thread is designed to contain all of the basic (non-virtual memory) information necessary for managing a thread. Kernel thread Alphabit (i.e., the KTCB) contains space for storing the processor's registers when a thread is descheduled, it keeps copies of the thread's user and supervisor stack pointers, and has provisions for being linked into different lists (e.g., the Ready Queue or a

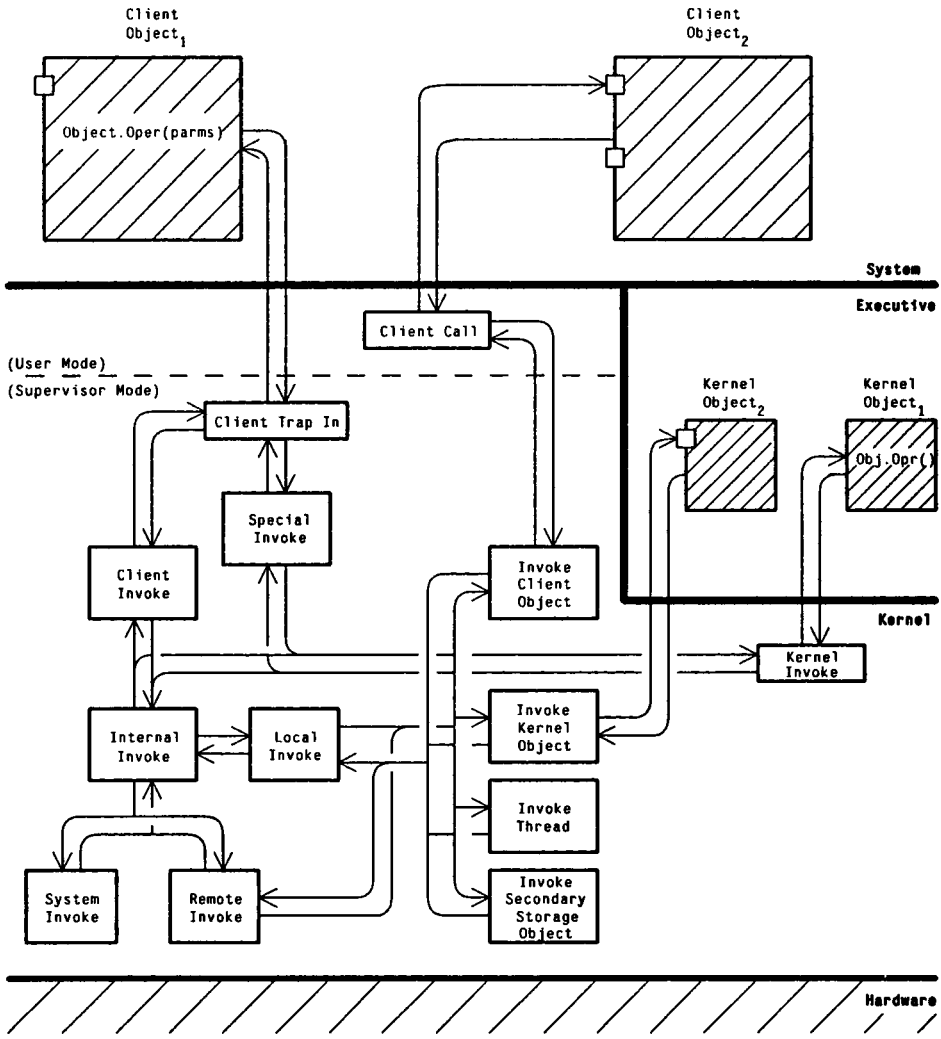

Figure 5-13: The Operation Invocation Facility

semaphore's blocked thread list). Furthermore, the kernel thread Alphabit contains the thread's environment information, that consists of the information necessary for managing threads within nested atomic transactions, and the information needed by the scheduling processor to perform the desired thread scheduling function. The KTCB is a structure very much like a standard process control block.

Each client thread in the kernel is constructed from a kernel thread. Each client thread Alphabit is bound to a kernel thread Alphabit (i.e., the CTCB and KTCB are linked together with pointers). Each client thread has a context associated with it, and when an object is mapped into the thread's context, it appears similar to the standard implementation of processes.

A CTCB contains all the necessary virtual memory information for the client thread, an indication of which PMC slot the thread is currently mapped into, and a pointer to the Alphabit of the currently mapped client object. When executing within the kernel each client thread uses its own kernel stack. The kernel's implementation is simplified by having each thread be able to block on its own stack, as opposed having a communal stack shared among all of the threads currently in the kernel.

Both types of threads may be involved in the invocation of operations on both types of objects (i.e., kernel and client). Each of the four possible combinations have different requirements and characteristics. If a kernel thread invokes an operation on a kernel object, no virtual memory manipulations are required — this type of invocation is the simplest and the most efficient of the four cases. If a client thread makes an invocation on a kernel object, no virtual memory manipulations are required, the client thread just traps from the client thread's context into the kernel — this too is a simple and efficient activity. If a client thread invokes an operation on a client object, the kernel is entered, virtual memory mappings are manipulated to remove the current object from the thread's context, the invoked object is mapped into the thread's context, and the client thread's context is reentered — this is the normal case supported by the kernel, and is moderately complex and moderately costly. If, however, a kernel thread invokes an operation on a client object, a substantial amount of activity has to take place in order to provide the kernel thread with the context necessary to perform an invocation on a client object. In order for a kernel thread to invoke an operation on a client object, it must (temporarily) become a client thread. This activity is straightforward in Alpha, because the design of threads is such that each client thread begins with a kernel thread and assumes the necessary control structures to become a client thread as part of its initialization.

To become a client thread, a kernel thread acquires and initializes a full set of virtual memory management data structures, acquires and initializes a context to execute in, maps into this context the destination object, then enters and begins execution just as if it were a client thread's invocation of a client object. Once the invoked operation on a client object is complete, the client thread's data structures and context are deallocated, and the execution of the kernel thread continues. This type of invocation is the least common, as well as the most complex and costly, of all the types of invocations.

The kernel-provided atomic transaction and deadline constructs are block structured. The object programming language pre-processor enforces the restriction that the constructs that mark the beginning and end of an atomic transaction or a collection of statements with a deadline must occur within the same operation within an object. Whenever such a block structured construct is aborted (i.e., when a transaction aborts, or a hard deadline cannot be met), the kernel must skip the execution of the thread forward to the instruction following the end of the aborted block. To accomplish this, the kernel maintains a (stack-oriented) collection of block return points, along with the nesting information kept in the KTCB's. When a thread aborts a block for whatever reason, this information is used to ensure that the thread is returned to the proper place of execution and the thread is prepared to abort to the next higher block (if it was nested). The same mechanism is used to abort transactions, abort portions of broken threads, and abort computations with hard deadlines (i.e., the computation has no value if its deadline cannot be met).

5.4.4. Access Control

In some systems (e.g., [Wulf 81] and [Jones 79]), capabilities have a set of *rights* that define the types of operations associated with each capability. In Alpha, the converse is true in that *restrictions* accompany a capability — if a capability is held by an object, that object may perform any operation that is not specifically prohibited by the restrictions associated with the capability.

The manifestations of capabilities local to an object are mapped by the kernel into their internal representations known as *Capability Descriptors*. A Capability Descriptor is composed of two components: an object's Alphabit Identifier and the set of restrictions placed on the capability's use. The restrictions associated with a capability are represented by a *Restriction Vector* containing the per-operation restriction specifications.

An object's c-list consists of an array of Capability Descriptors, located in each object's control block, managed as a random access list. Since the c-list is part of the object control block, it can be accessed only by the kernel (executing in supervisor mode in the kernel context). Objects do not directly manipulate capabilities, but rather, access their capabilities by way of indices relative to their c-lists. Currently, c-list indices are assigned in sequential, ascending order, although some type of encoding scheme may be employed in the future to reduce the probability of errantly using a valid c-list index.

On operation invocation, capability validation and translation is performed as soon as possible; the destination object capability is checked first, following that, any other capabilities in the parameter list are checked. On return, only the capabilities in the return parameter list must be checked.

To pass a capability as an operation invocation parameter, the kernel takes the c-list index for the destination object and any of the indices of capabilities passed as parameters, and then traps into the kernel. Once in the kernel, the operation invocation facility translates capability indices into their corresponding Capability Descriptors by looking them up in the client object's c-list. Each capability index is checked for validity during the translation process — i.e., that it points to a valid entry in the c-list. Next, the restrictions portion of the Capability Descriptor is checked to verify that the invoking object has the right to use the capability in the requested manner (i.e., invoke an operation on the destination object, copy the capability, transfer the capability, etc.). Should the validation process fail on any capability, the whole invocation (or invocation return) fails, and a failure indication is passed back to the invoking object. When an invocation fails due to the use of invalid capabilities, the invocation facility does not pass any parameters other than the error indication.

In the course of capability validation, it is determined whether the Capability Descriptor should be modified. If the capability may only be used once, or if it is being transferred without keeping a copy, the capability is removed after it is validated and translated. If a Capability Descriptor has had restrictions applied to it to the point of making it unusable, the entire capability is removed from the object's c-list.

As capabilities are passed to invoked objects, the invoked object accumulates the least restrictive capabilities for a given object. In particular, if an object receives a Capability Descriptor for an object for which it already possesses a capability, the Restriction Vectors of the two Capability Descriptors are compared, and for each restriction on each operation, if either of the vectors does not have the restriction, then the resultant vector does not have the restriction. However, if an invoked object receives a Capability Descriptor for an object for which it does not already possess a Capability Descriptor, a new capability is added to the object's c-list. This is accomplished by performing a logical OR function on the Restriction Vectors.

When an object is created, a capability is given to the creating object that has only those restrictions which are specified in the object type specification. While capabilities can not be amplified (i.e., restrictions cannot be removed), restrictions may be added to a capabilities when they are passed as parameters in operation invocations. This is done by applying the **RESTRICT** language construct to capabilities in an invocation parameter list. This construct allows the specification of the restrictions that should be added to a particular capability. For each usage of the **RESTRICT** construct, the object language pre-processor generates a mask that is passed with the parameters to the operation invocation facility and is used in the translation process to modify the Restrictions Vector of a Capability Descriptor before passing it to the destination object.

5.4.5. Concurrency Control

The Alpha kernel provides a set of kernel-defined objects that support the kernel's concurrency control mechanisms — semaphore objects and lock objects.

5.4.5.1. Semaphore Mechanism

The creation of an instance of a ***Semaphore*** system service object involves the allocation and initialization of a semaphore data structure attached to the invoking object's Alphabit. A semaphore data structure consists of a variable that contains the semaphore's current count, and a pointer to the list of KTCB's that are currently blocked waiting on the semaphore. The kernel takes advantage of the small amount of data required by a semaphore data structure and statically allocates a fixed number of semaphores for each object.

When a new instance of a semaphore is created, an unused semaphore data structure is initialized (with the count given as a parameter to the create operation) and a capability is returned to the invoking object. A create operation on the ***SemaphoreManager*** object returns a failure indication if all of the object's semaphore instances have been used.

Because ***Semaphore*** objects are permanently bound to the object that created them, not only can the semaphore data structures be associated with the object's Alphabit, but a special type of capability can be used. To an object, the capabilities for semaphores appear the same as any other ones (i.e., a c-list index), but the Capability Descriptor does not contain the standard Alphabit Identifier and Restrictions Vector, but rather it contains an indication of the special nature of the capability and an index to the ***Semaphore*** object's data structure.

A semaphore data structure contains a count field (manipulated by the operations defined on semaphores), a list of the threads that are blocked on the semaphore (waiting for its count to become greater than zero), and a list of references to the threads that have issued `P` operations on the semaphore and have not yet issued matching `V` operations. A `P` operation decrements the semaphore's count, and should the count become nonpositive, the invoking thread is blocked and added to the semaphore's list of blocked threads. A `V` operation increments the semaphore's count, and should any threads be blocked on the semaphore at this time, a thread from the semaphore's list is unblocked. The `C_P` operation first tests if decrementing the semaphore's count would cause it to become nonpositive. If so, the operation returns a parameter indicating this, otherwise the operation is performed as though it were a normal `P` operation.

Currently, the kernel manages the list of threads blocked on a semaphore in a FIFO fashion. However, semaphores were designed to operate in concert with the scheduling subsystem. This was done to allow the selection of appropriate thread to be unblocked to be performed according to the policy used in scheduling the application processor.

The kernel maintains, for each semaphore, references to the threads that have P operations outstanding on the semaphore. This is because the kernel must keep track of which threads have outstanding P operations on semaphores, in order to issue matching V operations should a thread abort (as a result of an aborted transaction, a missed hard-deadline, or as part of the thread repair function). In the course of cleaning up objects on a thread abort, the kernel must check the list of threads associated with each of the objects' semaphores, and for each occurrence of the thread being aborted in a semaphore's list, the kernel issues a V operation on that semaphore.

5.4.5.2. Lock Mechanism

Lock objects are similar to ***Semaphore*** objects in implementation, and instances of ***Lock*** system service objects are created and accessed in the same manner as ***Semaphore*** objects. However, each instance of a ***Lock*** object is associated with a region of data in an object, and each lock has a matching sized area of kernel memory that is used as a write-ahead-log, and is allocated from the kernel's heap region when the lock is instantiated. A collection of data structures similar to the ones used for ***Semaphore*** objects are associated with the object's Alphabit.

The ***Lock*** object data structure consists of a field that indicates the mode that the lock is currently in (if it is currently held), a pointer to its log area, and a pointer to the list of threads blocked while waiting to acquire the lock along with an indication of the mode each thread is attempting to acquire the lock in.

For simplicity in accessing and changing it, the Lock Compatibility Table is implemented as a bit array. When a compatible request is made the lock is granted the requester, the memory area associated with the lock is copied into its log area, the mode of the lock is changed, and the thread is permitted to proceed. If an incompatible lock request is made, the invoking thread's KTCB is linked into the list in the lock's data structure and the lock's request mode is registered with it. When a thread issues an UNLOCK operation on a ***Lock*** object, the lock mode is restored to its previous state and the blocked thread list is examined.

Like those of semaphores, lock data structures include a list of the threads that currently hold the lock

(in any mode) in order to allow the kernel to release all locks held by an aborted thread. The thread to be unblocked when an UNLOCK operation is invoked is chosen from the set of threads in the lock's block list. Currently the kernel selects the first thread with a compatible, outstanding request. However, as with semaphores, the kernel was designed to have the scheduling subsystem choose which of the threads waiting in a lock's blocked list should be unblocked.

5.5. Resource Management Daemons

In Alpha, a daemon is defined to be a kernel thread and kernel object pair that is provided by the system to perform a system function, independent of any higher-level (system or application) activities. As part of the executive layer of Alpha, there are a number of kernel threads and kernel objects that serve to assist in the management of various critical, system resources. The resources managed by daemons include physical memory pages and both client and kernel, thread and object Alphabits. Most of the daemons in Alpha serve the same function — i.e., to ensure the supply of a set of preallocated resources in an attempt to speed the expected time for their dynamic allocation and deallocation — and most of the daemons perform their function in a similar fashion.

For each critical resource within the kernel there are a set of routines (generically known as *create* and *delete*) that are used for the allocation and deallocation of the desired resources. The create routines perform the necessary initialization operations for the desired resource and use routines (known as *get* routines) to acquire the partially initialized, preallocated resources from a pool. Each type of critical resource also has a routine to return these partially initialized resources to their associated pool (these are known as *put* routines).

Should attempts be made to obtain a resource when its associated pool is empty, the requesting object does not wait until the daemon can run and replenish the pool. Instead, whenever pools are exhausted, resources are dynamically allocated via routines that construct them from their basic memory components (known as *alloc* routines). Also, these resources are not completely deallocated (i.e., returned to the kernel's dynamic storage allocator) when they are returned to their pools, but rather the daemons eliminate excess resources when there are too many in a pool. This is done with routines (known as *dealloc* routines), that return the resources to their elemental forms.

5.5.1. Physical Memory Management Daemon

The *page daemon* attempts to keep a small number of free pages of physical memory in the *free page pool* at all times. This daemon deals with pages of physical memory, and interacts with the secondary storage subsystem to perform its functions. Whenever the page daemon is activated, it executes the given victim selection algorithm to determine which page(s) should be written out to memory in order to replenish the pool.

The victim selection algorithm used in the current implementation is a simple FIFO-like replacement algorithm. To this end, the kernel maintains a circular list of currently allocated memory pages in order of their allocation. To select a victim, the page daemon begins the searching at the top of the allocated page list (i.e., starting with the page allocated the longest time in the past). The page daemon searches this list in order until it finds a page that does not have to be written out before being reused (e.g., one that is read-only or has not been modified), or until the search has progressed through N pages in the list. When one of these search termination conditions is met the page pointed to is chosen as a victim, and is then given to the secondary storage facility that is responsible for copying the victim page to secondary storage. The page daemon continues to select physical pages until enough pages have been selected to bring the free page pool up to its nominal level. The page daemon then blocks until the secondary storage facility indicates that the selected pages have been written to secondary storage, at which point the page daemon adds the pages to the free page pool and blocks again.

The page daemon contains the page replacement policy, and can be easily modified to take the characteristics of the threads that own a page into account when selecting a victim page. In this way, the victim selection algorithm's preference for pages that do not have to be written back to secondary storage could be augmented to also prefer pages of threads with less value to the application. The page replacement algorithm can be based on the same thread environment information (e.g., urgency, importance, or accumulated computation time) used by the scheduling facility.

The upper limit associated with the free page pool is not meaningful, as the number of pages in it can never be too large. Thus, the page daemon is unblocked only when the pool's lower limit is crossed. Furthermore, the secondary storage subsystem is responsible for performing any of the optimizations that might be used to increase paging performance (e.g., caching, managing the physical layout of the disks, or reordering the execution of access requests).

5.5.2. Object and Thread Management Daemons

There are four daemons dedicated to managing the pools of preallocated data structures for kernel and client, threads and objects. The resources contained in the pools are known as *skeletons*, and consist of generic Alphabits and their associated data structures. Skeletons are initialized with the generic information for each of the types of data structures. When a skeleton is removed from a pool and put into use, the specific information relating to the individual instances of the use of a skeleton is added to it.

These skeletons consist primarily of Alphabit control blocks, and in the case of client threads or objects, virtual memory information (i.e., Extent Descriptors and PMAPs). The major savings in using the preallocated resources derive from not having to dynamically allocate the physical memory space for all of the data structures and from not having to fill in the various fields of the control blocks. In the case of client threads or objects, the saving is somewhat greater in that Extent Descriptors need not be obtained and initialized with segment map fields.

- MULTIBUS is a trademark of Intel Corporation.
- DVMA is a trademark of Sun Microsystems.
- Ethernet is a trademark of Xerox Corporation.

6
Hardware

The Archons Testbed is the hardware base on which the Alpha kernel is constructed. This facility was constructed following a study of the needs of the Archons project and an extensive survey of commercially available hardware [Clark 83]. This testbed is composed of a collection of hardware that meets the project's minimum functional specifications, conforms to the project's cost constraints, and required a minimum amount of custom hardware fabrication to put it in place. This description is of the current testbed configuration — however, there are plans for a series of upgrades to the testbed (including the use of next generation processors, semiconductor-based secondary storage devices, and a high-performance, real-time-oriented communication subnetwork).

The following describes the Alpha system hardware in three levels of detail — the *system* level, the *node* level, and the *processor* level. The system level encompasses the distributed computer on which the kernel runs, the application being controlled, and the hardware used for control and development work. The node level involves the architecture of the individual processing nodes in the distributed computer system. The processor level defines the components and structure of the processing elements in each node.

6.1. System Level Structure

The system hardware context in which the Alpha kernel has been developed consists of three main parts — the *application subsystem*, the *distributed computer subsystem*, and the *control subsystem*.

6.1.1. Application Subsystem

From the point of view of the Alpha kernel, the application subsystem is comprised of a set of *application devices*. The application subsystem is the interface between the distributed computer on which the Alpha kernel executes, and the physical system it is controlling. Application devices can be sources of data to the computer (e.g., sensors) or data sinks (e.g., actuators). Because the application

domain of Alpha is real-time command and control, these devices are considered to be intelligent, in the sense that all highly repetitive, fast response-time (i.e., sampled-data loop) processing is expected to be handled by the device interface controllers. Low-level signal data is therefore expected to be pre-processed and not passed around in bulk form on the distributed computer's communication subnetwork. This is not the case in low-level process control or signal processing applications.

Each of the application devices is connected to the distributed computer either at a node, or directly to the communication subnetwork. Connecting an application device directly to the distributed computer's subnetwork is equivalent to placing a peripheral controller into a conventional uniprocessor's backplane. Devices that are connected in this fashion must be capable of either simulating an object interface, or executing the communication protocols necessary to interact with an object on an Alpha processing node. Alternatively, devices are attached to one or more of the nodes in the system, which are then responsible for performing the necessary object encapsulation of the devices.

The application subsystem in Alpha is composed of a collection of processing elements that execute software to simulate the actual physical process being controlled. These processing elements are attached directly to the testbed's communication subnetwork.

6.1.2. Distributed Computer Subsystem

The distributed computer subsystem provided by the Archons testbed consists of a set of processing *nodes* interconnected by a global communications subnetwork. The nodes in the system are the individual hardware computational units on which the Alpha kernel executes. Nodes in the current testbed are homogeneous in architecture as well as in their interconnection to the global communication subnetwork.

The Archons Testbed consists of a set of (currently six) custom-built shared-memory multiprocessor nodes, based on Sun Microsystems Model 2/120 workstations. Each node consists of a collection of (3 or more) processors with local memory, shared memory, and various I/O device controllers (e.g., serial-line, network, or disk), all in a common MULTIBUS backplane (having 9 slots with specialized P2-bus connections).

The nodes are interconnected by a private Ethernet cable that is connected to the Computer Science Department's Ethernet via a gateway. The gateway machine is a standard Sun Microsystems workstation, running the UNIX operating system, with an Ethernet controller connected to each of the Ethernets, and a 16 channel asynchronous serial line multiplexer. The gateway machine runs

software that allows it to selectively connect or disconnect the two networks, and to communicate with the serial line channels connected to the console ports of the processors in each node.

6.1.3. Control Subsystem

Although it is possible that the application system itself may contain all of the devices necessary for interacting with and controlling this experimental system, this is generally not presumed to be the case. Therefore, a set of *control stations* is integrated into the system for the purpose of development, control, and monitoring of the Alpha system. In the current prototype, the control stations are standard Sun Microsystems workstations, running the UNIX operating system. The control stations interface with the Alpha nodes via the 9600 baud serial lines provided by the gateway machine.

To support the remote development of standalone operating system code, a number of software tools were created, modified, or simply exploited in order to provide an effective development environment. To this end, the Sun Microsystems window management tools are used to allow the developers simultaneously control multiple testbed nodes. Also, users of the testbed are provided with the ability to control the AC power to the individual testbed nodes (to clear hung nodes and for fault-tolerance experiments). The degree of control over the testbed nodes is so great as to permit the development of low-level system code in a totally remote fashion.

Figure 6-1 is a representation of the current Archons Testbed. This diagram illustrates the control and development workstations as well as the testbed nodes, but does not show the processing elements comprising the application subsystem.

6.2. Individual Node Structure

The Archons testbed provides a distributed computer system, composed of a collection of processing nodes, and interconnected by a global communications subnetwork. These testbed nodes were designed to support the project's goal of raising the semantic level of traditional system device interfaces. The objectives of this goal are to reduce the semantic gap between the operating system abstractions and the underlying hardware, to achieve concurrency within the operating system itself, and to explore the possibilities of achieving increases in system performance through the downward migration of functionality. The nodes of the testbed support this goal by providing the system-builder with a collection of general-purpose processing elements that can be used to provide the desired subsystem functionality without having to construct custom hardware.

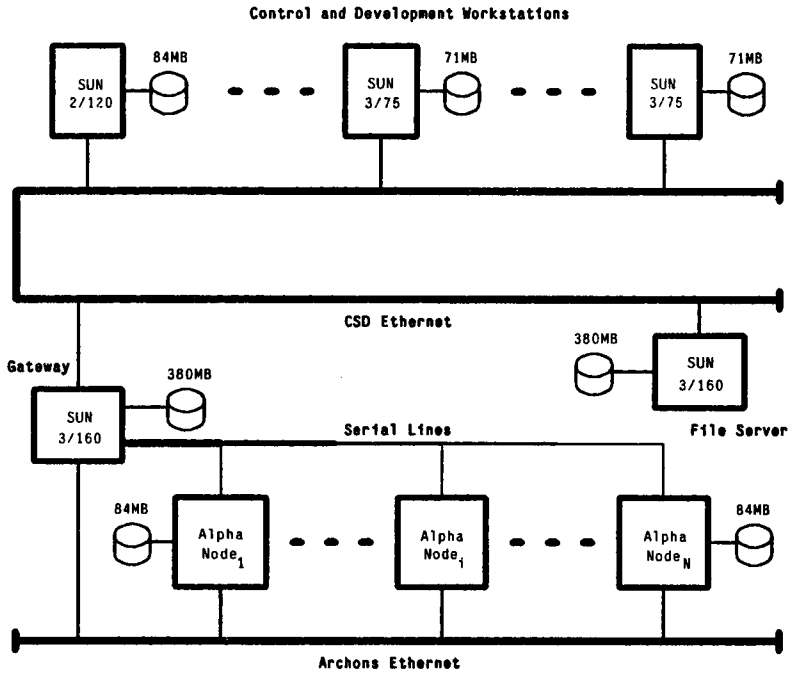

Figure 6-1: The Current Archons Testbed Facility

In support of these objectives, each node in the testbed is a symmetrical, shared-memory multiprocessor that is used to emulate a uniprocessor with intelligent device controllers by having one processing element (the *applications processing element*) be dedicated primarily to application programs, and as much of the system software as possible supported by peripheral control subsystems. A node's peripheral subsystems are, conceptually, intelligent device controllers, that are implemented by general-purpose processing elements (similar to the applications processing element), accompanied by standard device controllers. Each processing element in a node shares common memory via the node's common backplane, and has its own, on-board devices and local memory. Processing elements in a node typically execute out of their common memory, and interact with each other via shared memory and an interprocessor interrupt mechanism[2].

Ideally, the application software executes alone in the application processor. However, because of the

[2]The interprocessor communication mechanism is a custom designed and implemented hardware module which is added to the applications processor and is equally accessible to all of the processing elements in a node

current limitations of the testbed, the application processing elements each execute a copy of the Alpha kernel in addition to the application code. Ultimately, the hope is to migrate all of the system functionality into operating system processing elements and intelligent peripheral controllers. In such a system configuration, there might be multiple applications processing elements at each node, functioning in a manner similar to floating point co-processors — the application processing elements execute only the application code given to them by the system processing elements.

A representative processing node (as depicted in Figure 6-2) includes an applications processing element, a scheduling processing element, a communications processing element, a secondary storage processing element, and a common MULTIBUS memory unit. The applications processing element is a Sun Microsystem's version 2.0 processor, with (a minimum of) 1MB of local memory and an inter-processor interrupt module. The peripheral processing elements are (for reasons of cost and availability) Sun Microsystems version 1.5 processor boards, with custom hardware and firmware modifications. The Sun 1.5 processor board is a discontinued product of Sun Microsystems that was based on the original Stanford University MC68000 single board computer design, and modified to use the MC68010 microprocessor. The communications processing element has associated with it a Sun Microsystems Ethernet controller. This board consists of an Intel 82586 Ethernet controller chip, 256KB of local buffer memory, and an address translation unit. Likewise, the secondary storage processing element has associated with it a high-performance disk controller and 84MB disk (with an SMD interface).

6.3. Processing Element Structure

At present, there are two different versions of the same processor design being used in the processing elements of each testbed node. In Alpha, one type of processor is used in the applications processing element, and the other is used in the peripheral processing elements. The following is a brief description of the major features of each of the two types of processors.

6.3.1. Peripheral Processing Element

In the current testbed the processors used for peripheral processing elements in each node are modified versions of the Sun Microsystems version 1.5 processor board. This processor is a MULTIBUS single-board computer that is based on a 10MHz Motorola MC68010 microprocessor, with one wait-state access to 256KB of on-board (and up to 2MB of off-board) read/write memory (with byte-parity) and 32KB of programmable read-only memory. This processor also contains a Memory Management Unit (MMU) that supports a segmentation-with-paging virtual memory management

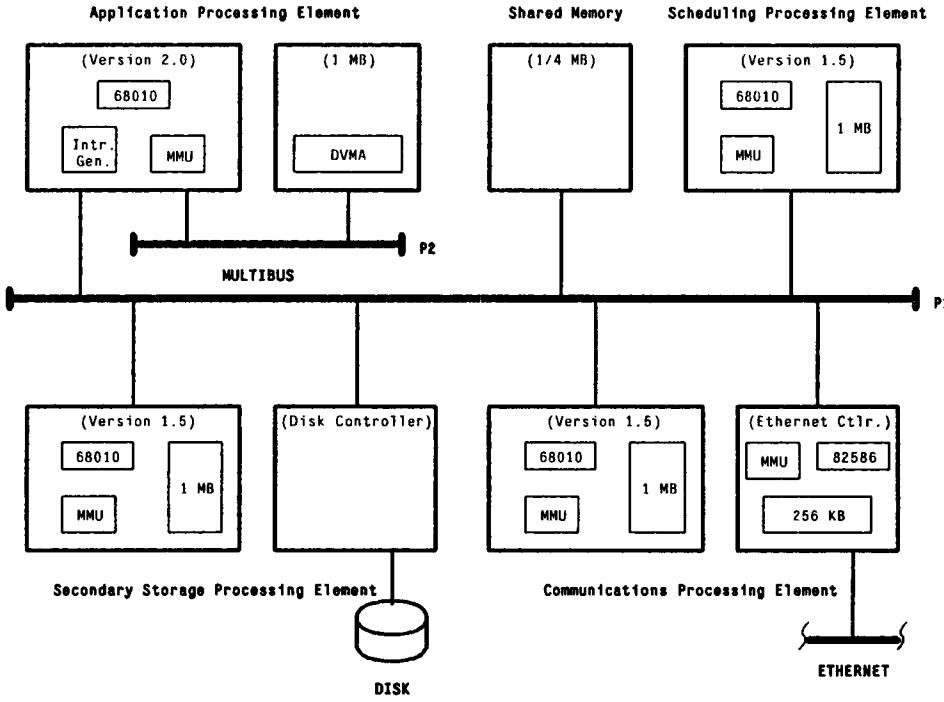

Figure 6-2: The Structure of an Alpha Processing Node

scheme. Also included on the processor board is a dual independently programmable USART, five programmable 16-bit timers, and a 16-bit parallel input port. Furthermore, the Sun Microsystems 1.5 processor supports MULTIBUS multimaster arbitration, and allows the expansion of on-board memory via a proprietary definition of the MULTIBUS P2 connector.

The structure of a Sun Microsystems 1.5 processor board is depicted in Figure 6-3. Most of the features of the CPU are standard and need little explanation. The Motorola MC68010 is a multi-register, stack-oriented microprocessor, supporting a linear virtual address space of up to 16MB and having both client and supervisor protection modes, with a separate stack pointer for each mode.

The modifications that were made to these processors include: configuring the processor to function properly in a multimaster MULTIBUS environment, reassigning the timers in order to cascade a pair of timers for 32-bit operation, upgrading on-board memory to 1MB, and adding the Arpa Internet Trivial File Transfer Protocol (TFTP) and a serial-line (Motorola S-record) downloading program to the PROMs.

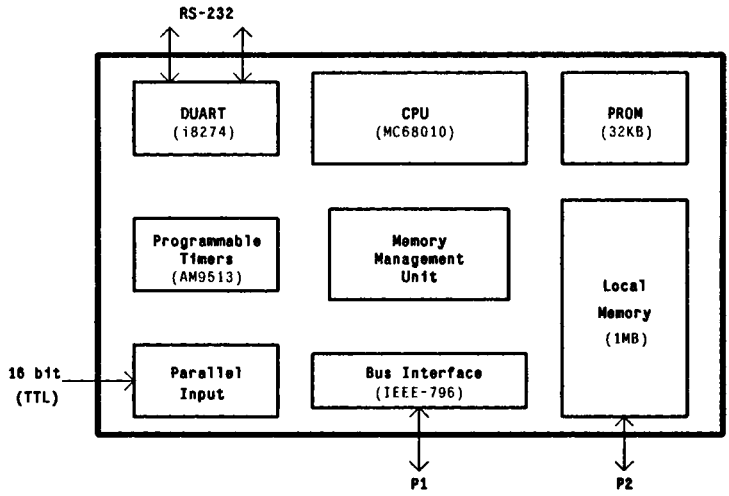

Figure 6-3: Sun Microsystems Version 1.5 Processor Board

While the MC68010 CPU does not provide memory management directly, it does support instruction restart to permit the implementation of demand-paged virtual memory, and the Sun Microsystems version 1.5 processor has a custom MMU on-board. The version 1.5 MMU supports a linear virtual address space of 2MB, which is divided into 64 segments (each consisting of a maximum of 32KB of memory) which are further divided into pages of 2KB each. The MMU is implemented as a two-level translation table and therefore has fixed limitations on the number of segment and page table entries. A memory location known as the *context register* defines the current address mappings for the processor by selecting one of the 16 sets of segment table entries (known as *segment maps*). Each segment map entry consists of an index to a *page map*, consisting of 16 page map entries, each of which may contain the physical address of a single 2KB page. The page map table contains 64 page maps, each of which represents one segment of 32KB of virtual address space. The structure of the Sun Microsystems 1.5 MMU is depicted in Figure 6-4.

A virtual address is translated to a physical address in the following way. First, the context register is used to determine the appropriate segment map. The first 6 bits of the virtual address are then used to index into this map. This yields an index into the page map cache, thus indicating the appropriate page map table. The next 4 bits of the virtual address then select the appropriate page entry within the page map table. This yields a physical page index, and the final 11 bits of the virtual address are used to select the correct byte within the physical page.

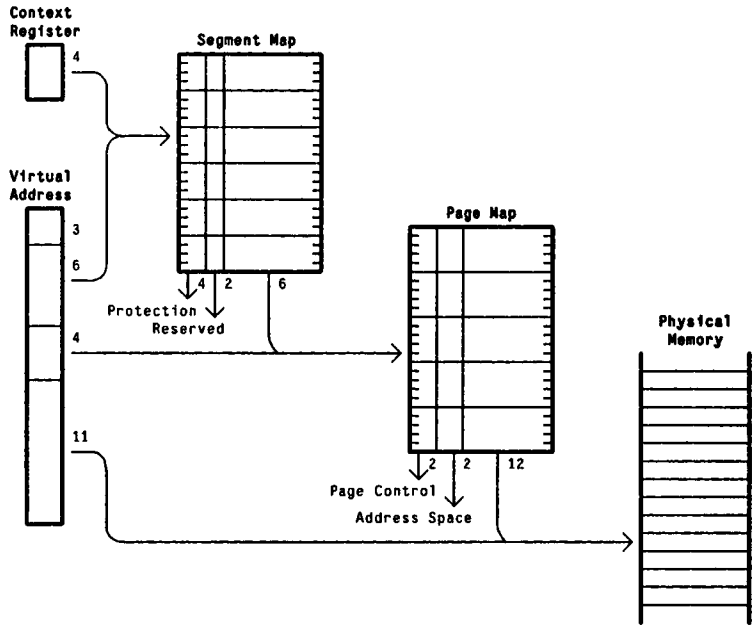

Figure 6-4: Sun Microsystems Version 1.5 Memory Management Unit

It is possible that a *fault* may occur during translation of a virtual address. This can happen because a page map table has not been allocated to the currently executing context (i.e., a *segment* fault), or because no physical page is allocated at the referenced virtual address (i.e., a *page* fault). In these cases, kernel software must determine whether a demand allocation is called for, and the faulting instruction is to be continued from the point of the fault, or whether the fault signals a system error and the instruction should be aborted.

In the Sun Microsystems 1.5 processor, the MMU provides protection only on the segment level, and the protection bits are not fully decoded (i.e., not all read/write/execute operations can be performed in both superviser and user modes). However the version 1.5 MMU does provide per-page statistics for both page reference and page modification.

The Sun Microsystems 1.0 processor board used the MC68000 microprocessor, and its MMU was implemented in such a manner as not to introduce any wait-states to the local memory access time. However, timing changes in the MC68010 microprocessor introduced the need for one wait-state in

each memory access on the version 1.5 processor board. Further details of these Sun Microsystems processors and their MMUs can be found in [Bechtolsheim 82], [Sun 82], and [Sun 84].

6.3.2. Application Processing Element

The Sun Microsystems version 2.0 processor board, which is used as an applications processor in the testbed, is essentially the same as the version 1.5 processor, and so only the differences are outlined here. The general structure of the version 2.0 processor (as shown in Figure 6-5) is quite similar to that of the version 1.5 processor.

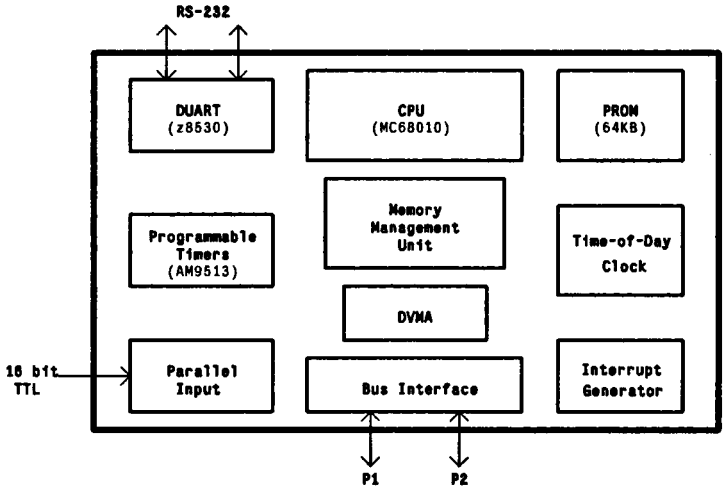

Figure 6-5: Sun Microsystems Version 2.0 Processor Board

The version 2.0 processor: has no on-board memory, can access primary memory without wait-states, can have as much as 8MB of physical (local) memory, provides a full 16MB of virtual address space per context, performs memory refresh in hardware, allows access to its primary memory via Direct Virtual Memory Access (DVMA) from the MULTIBUS. In addition, the USART was changed to a Zilog 8530, and a battery backed-up time-of-day clock was added to the Sun version 2.0 processor board.

Perhaps the most significant changes in the 2.0 processor board has to do with the MMU. The Sun 2.0 MMU has a larger number of both segment and page table entries, protection is now performed

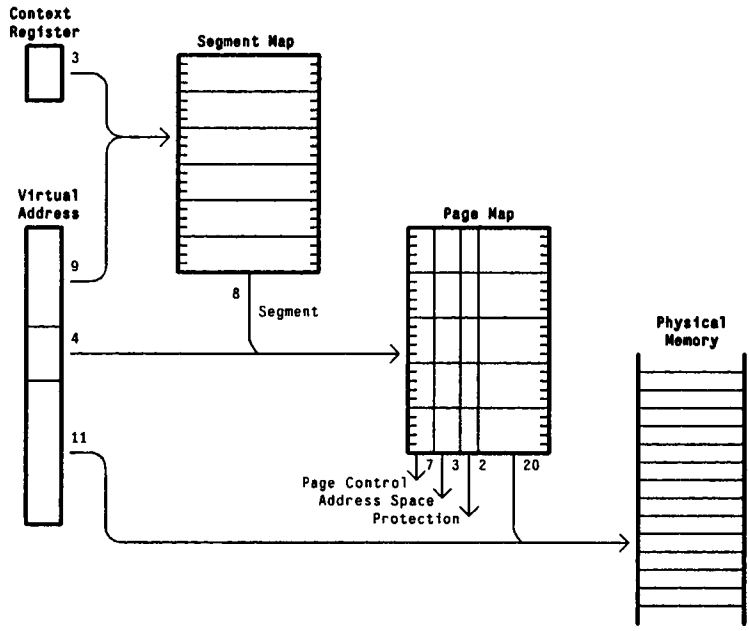

Figure 6-6: Sun Microsystems Version 2.0 Memory Management Unit

on a per-page level, and the protection field in the page descriptor is fully decoded. The CPU's function codes are used to support multiple, full 16MB address spaces, including: primary memory space, on-board I/O space, MULTIBUS memory space, and MULTIBUS I/O space. Furthermore, only 8 contexts can be loaded in the Segment Map Cache at any one time. The structure of the Sun Microsystems 2.0 MMU is shown in Figure 6-6.

6.4. Hardware Implications

A significant contribution of this work is the illustration of how a logically integrated system can be constructed on physically distributed hardware. This research contributes to the existing body of knowledge by describing those architectural features that proved beneficial (and those that were detrimental) in implementing the kind of programming abstractions for the class of system considered here.

In the course of this work it has become apparent that many of the low-level hardware issues, frequently dismissed as insignificant implementation details are in fact quite significant to the overall

quality of the system. Clearly the design and implementation of an operating system are strongly influenced by the underlying hardware. Not so obvious, however, is the fact that the overall practicality of a system may depend on the nature of the hardware and its effective use.

The implementation of the Alpha kernel described in this document is the second incarnation of the kernel. The first system was built on nodes that used the Sun Microsystems version 1.5 processor as applications processors, whereas the implementation described in the previous chapters is based on a version 2.0 applications processor. Given the great similarity between the two processors, it seems unlikely that there would be a great difference in the design of the kernel for the two processors. However, this is a graphic example of how "engineering details" have a dramatic effect on the overall structure and effectiveness of a system. The initial hardware led to a number of design decisions that had severe negative impacts on the kernel, that only became apparent after the first version of the system was built.

6.4.1. Problems with the Initial Processor

The features of the version 1.5 processor that had detrimental effects on the initial design of the kernel are related to: the small virtual address space, the design of the MMU, the lack of support for multiprocessor implementations, the use of software memory refresh, and inadequate timer support.

— Restricted Virtual Address Space Size

The most regretted decision made in the initial design resulted from the (perceived) virtual address space restrictions of the version 1.5 processor. The fact that a major portion of the virtual address space is used to provide memory-mapped access to the MMU and peripheral devices on the application processor board results in a usable virtual address space of only 2MB per context. While this size might be considered adequate for most client contexts, it was thought that such an addressing limitation would not properly allow the kernel to coexist with a thread and object in each client context.

The decision was therefore made to have the kernel exist in a context separate from that of the client threads. Each client context was designed to share a common segment (known as the *context exchange region*) through which all access to the kernel context is mediated. This constraint imposes a cost that recurs each time the kernel is entered and exited (which occurs four different times in the course of a normal operation invocation). Furthermore, some information was required to be accessible in both the client and kernel contexts. This introduced the additional overhead of adjusting references to parameter pages and the kernel stacks of threads on entering and exiting the kernel.

Thus, as a result of the using the processor's physical address lines to address the on-board devices, a significant performance penalty was incurred upon each operation invocation performed by the kernel and the entire structure of the kernel was affected.

— Memory Management Unit Design

Another feature of the version 1.5 processor that had a significant effect on the design of the kernel is the design of its MMU. The version 1.5 processor's MMU was designed to provide protection only at the segment level. This does not allow the fine degree of protection control that is needed for doing various per-page virtual memory operations (e.g., copy-on-write pages). The fact that protection in this processor is on a segment (as opposed to a page) level makes for a number of difficulties, such as: inhibiting the ability to protect code and data that may be combined into a single segment, being unable to adequately protect low-core, and not being able to do efficient, page-level protection of such structures as client stacks within thread contexts.

The segment descriptor protection field in the 1.5 processor's MMU is encoded, limiting the possible protection modes and requiring the explicit saving and restoring of the protection field if the segment should be temporarily invalidated. Another problem with the 1.5 MMU has to do with the fact that segment descriptors are not validated on page map cache accesses. This means that additional kernel complexity is introduced to deal with the fact that a segment fault will not occur if a segment descriptor is invalidated between validating a segment descriptor and accessing one of its page descriptors. Furthermore, the small number of page descriptors that can be loaded into the 1.5 MMU's page map table introduces another limited resource whose efficient management is critical to the overall system performance.

The version 1.5 MMU also does not allow the processor to access page map or segment map entries that belong to another context. This means that all such manipulations must be performed from within the target context, requiring that the cost of entering and exiting another context be paid each time such a manipulation is to be performed. In the case where large portions of a context are to be remapped from within another context, the cost of performing context switches is considerable. Also, because the version 1.5 processor board periodically asserts the MC68010's non-maskable interrupt, the kernel must ensure that the system stack pointer refers to a valid page at all times. Thus, each context swap must be accompanied by the movement of the system stack into the destination context.

— Support for Multiprocessor Configurations

Some complications in the initial design of Alpha resulted from the fact that the testbed processors were not designed to function in a multiprocessor context. The version 1.5 processor does not allow direct access to its memory, MMU, or on-board devices from the MULTIBUS. With the Sun Microsystems version 1.5 processor, information to be exchanged with the other hardware functional units in a node must be placed in shared MULTIBUS memory or explicitly moved by the application processor. This introduced the need for additional copying of information between the processors in a node, incurring performance degradation with each each access to shared MULTIBUS memory, as well as the additional complexity of having to jointly manage a limited amount of non-local memory among the processors in the node.

A further effect of the fact that the application processor was not designed for multiprocessor use is the lack of a means for the generation of MULTIBUS interrupts. While the processor does respond to bus-vectored interrupts on the MULTIBUS, it does not include a mechanism for generating them. Without this function, the processors in a node must communicate via the polling of shared memory locations — a technique that is more costly and less responsive than an interrupt-based interface.

— Software Memory Refresh

The version 1.5 processor implements the refresh of its local memory in software, as opposed to the more traditional use of transparent hardware to refresh dynamic memory. While this function in itself does not consume many processor cycles, the indirect costs of software memory refresh were significant. The software memory refresh function required that a non-maskable interrupt be generated periodically, and the ever-present nature of this interrupt incurred a cost in terms of both complexity and performance.

Because non-maskable interrupts are guaranteed to be generated periodically, a number of precautions had to be taken in the initial kernel code. These precautions imposed a fixed overhead cost in order to deal with an infrequent event. Each time the processor's context is changed, special care must be taken to ensure that the system stack pointer will be valid in the destination context. This is because interrupts require a valid system stack, or the processor board will enter a *hung* state that can only be terminated by power-cycling the processor. Thus, the ability to disable all interrupts for a short period of time would have proved to be a very desirable function.

Despite the fact that the software refresh of local memory costs very little in terms of processor cycles,

the potentially large hidden costs associated with this technique make it a poor design decision. While the cost of doing the actual refresh is small (i.e., 128 NOP instructions every 4 milliseconds), it incurs a significant cost as a result of synchronization demands imposed on the kernel (i.e., approximately 200 microseconds on each context switch or operation invocation).

With a small amount of additional hardware it would be possible to: provide hardware refresh of the local dynamic memory, limit the use of the processor's non-maskable interrupt to such functions as power-fail or external reset, and simplify the internal structure of the Alpha kernel.

— Programmable Timers

Finally, the programmable timer used on the version 1.5 processor has a number of features that make it difficult to deal with it. The design of this timer is such that reading the state of a counter within some window of time (70 nanoseconds) of a clock pulse causes undefined data to be read. Given that the processor and the Alpha kernel both clock these timers at high speeds (200 nanoseconds), this may occur with great frequency. This requires that all timer read operations must disable the clock, read its value, and then re-enable, which both adds to the cost of timer manipulations and lessens the accuracy of the timer by allowing clock pulses to be missed. Also, the internal registers of this timer are multiplexed into a two word window by selecting the desired registers in one command and manipulating the selected register in another command. However, because interrupts may occur between the selection and access operations, great care must be taken in the manipulation of the timer, thus adding further to the overhead of performing timer operations.

The version 1.5 processor uses a separate timer for each of the two USART channels, thus leaving only one timer (out of five) for the user. This does not provide adequate hardware timer support for the timed event and performance monitoring functions in Alpha. This required that the user timer be shared among all of the diverse functions in the kernel, at costs in complexity and performance.

Additionally, the lack of external reset pin on the timer makes it impossible to perform a complete external hardware reset of the application processor. If a hardware reset is issued while the refresh timer is running, the processor enters a hung state and must be power-cycled to be reset.

6.4.2. Fixes in the Current Processor

Most of the problems with the version 1.5 hardware have been dealt with in the Sun Microsystems version 2.0 processor board, including: the elimination of timing problems, allowing zero-wait state access to a maximum of 8MB of physical memory; the use of function codes and an increased number of segments per context, to provide a full 16MB virtual address space for each context; the use of hardware memory refresh, thereby eliminating the need for periodic non-maskable interrupts; the inclusion of fully decoded, page-level protection; and the support of DVMA, allowing arbitrary segments of the processor's virtual address space to be made available to MULTIBUS masters.

The only drawbacks of the Sun Microsystems 2.0 processor are that there is no on-board memory (additional boards must be connected via the processor's MULTIBUS P2 connector), and that only 8 contexts can be mapped in the 2.0 MMU at any one time (as opposed to 16 in the 1.5 MMU).

6.4.3. Suggested Processor Modifications

Although many of the negative aspects of the Sun Microsystems version 1.5 processor board have been corrected in the Sun Microsystems version 2.0 processor board, not all of the desired features have been incorporated into that design.

— Object-Oriented Memory Management Unit

Both of the Sun Microsystems MMUs suffer from the fact that they are not hardware translation caches, but rather a set of hardware lookup tables that must be made to function as a cache by the kernel software. Greater system performance and a reduction in the complexity of the virtual memory portion of the kernel could be attained through the use of a hardware address translation cache. Furthermore, the Sun Microsystems MMUs do not provide special support for the type of partial context swaps performed by the Alpha kernel on operation invocation. The kernel would benefit from an optimization that would allow the thread and object portions of a context to be switched independently.

An example of an MMU that is both a real hardware cache and allows the the kind of partial context swapping that would be useful in Alpha can be found in the Digital Equipment Corporation's VAX processors (with a translation lookaside buffer and the separate System, P0 and P1 regions). This type of MMU would incur a significantly greater cost in terms of hardware complexity, but would greatly reduce the complexity of the kernel software in the area of virtual memory management and would increase the performance of the system by decreasing the costs of operation invocations.

— Multiprocessor Interface

Both versions of the Sun Microsystems processor boards could be redesigned to perform better in the type of shared-memory multiprocessor configuration used as nodes in this type of system. For example, the use of an intra-node bus with an interprocessor mailbox mechanism (e.g., MULTIBUS II or VAX-BI) would be a useful addition to the processors in a node. In lieu of this, the ability of a processor to generate and clear MULTIBUS interrupts, and to mask (under software control) MULTIBUS interrupts would be useful.

These modifications would allow the application processor and the peripheral device processors that coexist in a node to interact in a symmetrical and high-performance manner. Blocks of information to be exchanged among the various components in a node could then be accessed directly and shared by altering memory mappings, as opposed to copying.

It is also important that all of the external memory references to the processor's local memory are made through the processor's MMU. Otherwise, a great deal of logical complexity would be introduced in order to deal with the fact that external references are made to physical locations, while internal references are made to virtual locations. Also, much of the benefit of sharing and moving blocks of memory by address mapping would be lost if the MMU were not in the memory addressing path for external references.

— Multiple Processor States

In order to properly support the kernel thread and kernel object approach to making extensions to the Alpha kernel, it is necessary to have more than the two states (i.e., user and supervisor) that the Motorola MC68010 microprocessor provides. This is because the kernel objects and threads execute in supervisor mode and need to provide stack protection and dynamic stack extension. If a fault were to occur on system stack accesses, there is no other state (with an associated stack) to handle the fault in.

— Protected Code Entry Points

Another function that is provided by the kernel software, and could be supported more effectively in hardware, is the protection of code entry points. The hardware could be modified to provide protection to the entry points of objects (similar to the *gate* concept found in the Intel 80286), which would provide a minor performance increase for operation invocations in Alpha.

6.4.4. Beneficial Processor Features

Despite the shortcomings of the initial hardware, there are a number of hardware features of both processors that proved to be quite useful in the design and implementation of the system, and some of these features contributed significantly to the success of the kernel. In general, the Sun processors, with a 10Mhz Motorola MC68010 microprocessor, custom MMU, and on-board USART and programmable timers, provide an outstanding environment for doing operating system development work.

The system power-on initialization and diagnostic code, contained in the on-board PROMs, help to make the processor more usable in a stand-alone context and to speed the development of the low-level kernel code. Furthermore, by providing sufficient PROM space, the processor allows the addition of a number of very large, highly capable programs to the processor's power-up state (e.g., the TFTP program).

Furthermore, by taking advantage of the fact that the Sun Microsystems processors have a local memory bus that is separate from the general-purpose, multi-master MULTIBUS, it is possible to construct multiprocessor nodes of Sun processors and other vendors' device controllers. The overall performance of the Alpha kernel would suffer if the application processor did not have a separate bus for local memory accesses, or if parallelism could not be achieved between the application processor and the peripheral functions.

The design of the Alpha kernel was heavily influenced by the fact that the testbed application processors support demand-paged virtual memory. By making use of the virtual memory support of the application processor board, separate addressing domains can be provided for threads, thereby supporting the system reliability goal of fault containment. Also, the use of the processor's virtual memory hardware increases overall system performance by minimizing the copying of information within the system, and permitting the use of such deferred-payment techniques as copy-on-write.

Additionally, the standard Sun Microsystems UNIX-based workstations provide an excellent development environment for the Sun processor-based standalone testbed.

- MULTIBUS and MULTIBUS II are trademarks of Intel Corporation.
- DVMA is a trademark of Sun Microsystems.
- VAX and VAX-BI are trademarks of Digital Equipment Corporation.

- UNIX is a trademark of AT&T Bell Laboratories.
- Ethernet is a trademark of Xerox Corporation.

7
Comparisons with Systems of Interest

This chapter describes a number of related operating systems and compares them to Alpha. The systems considered here include distributed systems that have actually been constructed; paper designs, simulations, and emulations of systems constructed from application code on top of existing operating systems are not considered interesting. The focus is also on systems that provide object-oriented programming interfaces and include reliability as one of their primary objectives. Explicitly excluded from the field of interest here are systems that consist only of run-time support for programming languages and highly specialized database systems. Because of the dearth of native operating system work in recent years, these constraints are not strictly adhered to in order to include some of the more interesting efforts that may not entirely fall within the specified domain of interest.

The systems chosen to be examined here are *Hydra* [Wulf 81], *StarOS* [Jones 79], *Cronus* [Schantz 85], *Eden* [Almes 85], *Argus* [Liskov 84], *Accent* [Rashid 81], and *Locus* [Popek 81]. A number of general observations can be made concerning this set of systems. None is concerned with real-time control[3], and none is both physically decentralized and logically centralized. *Argus* has been implemented on a centralized system (but work is underway on a distributed version of the system), *Hydra* and *StarOS* were built on multiprocessors, and the rest of these systems were constructed on a collection of machines interconnected by local area networks. Of all of these systems, only *Hydra*, *StarOS*, and *Accent* were built directly on bare hardware. The *Cronus*, *Eden*, *Argus*, and *Locus* systems are implemented on, in, or beside the UNIX operating system. All of these systems except *Cronus* and *Accent* had reliability as a major objective of their research. For purposes of reliability, the *Argus* and *Locus* systems support atomic transactions, while *Cronus*, *Eden*, and *Locus* support some form of replication. Of these systems, only *Accent* included a complete virtual memory facility as a part of the system. Most of these systems support an object-oriented programming paradigm, with the exception of *Locus* (which is a distributed UNIX system). Each system provides a different

[3] In fact, there is no publically available information on significant operating systems for distributed real-time command and control.

level of functionality — *StarOS* is an operating system, *Hydra* and *Accent* are kernels, *Cronus* is a constituent operating system (i.e., an operating system that coexists with other operating systems), *Eden* and *Argus* provide programming language interfaces, and *Locus* provides extensions to an existing operating system. All of these systems provide some form of location-transparent communication based on globally unique logical identifiers. The *Cronus*, *Accent*, and *Locus* systems involve low-level communications work, i.e., they do not (exclusively) use standard communication protocols or facilities.

7.1. Hydra

The *Hydra* operating system kernel was built at Carnegie-Mellon University by Bill Wulf, et. al., during the period from 1971 to 1981. *Hydra* was a capability-based, object-oriented operating system kernel that was constructed on the *C.mmp* multiprocessor [Wulf 81]. The *C.mmp* multiprocessor was constructed from standard processors and memories connected by a custom-built crossbar switch. *Hydra* was written largely in the BLISS-11 programming language. Applications were written for *Hydra* and experiments were performed on it until *C.mmp* was decommissioned in 1981.

7.1.1. Hardware

The *C.mmp* multiprocessor consisted of 16 (modified) PDP-11/40 and PDP-11/20 processors and 16 memory units (collectively) providing 2.6MB of shared memory, interconnected with a 16-by-16 crossbar switch. On each processor's UNIBUS, there was a small amount of local memory (i.e., 8KB primarily for the frequently used parts of the *Hydra* kernel), and all of the processor's peripheral devices. Some processors had disks for permanent storage and some had drums (i.e., fixed-head disks) for paging. In addition, each processor had UNIBUS interfaces to a global interprocessor interrupt structure, and a global time source. Both the UNIBUS and the crossbar switch allowed read-modify-write instructions to support kernel synchronization mechanisms. Furthermore, each processor's interface to the crossbar switch was connected to the UNIBUS and contained the address relocation information for that processor. The memory access times for all processors to the shared memories was uniform at 1usec per word, and the average instruction execution time of the PDP-11/40 is 2.5 microseconds.

7.1.2. Objectives

A major goal of the *Hydra* project was to explore the use of symmetric multiprocessors in a general-purpose timesharing environment. This effort hoped to achieve high degrees of performance, availability and extensibility, and to determine the manner in which contention for shared system resources restricts parallelism. The intent of this effort was to provide a kernel, without a file system, command language, or (long-term) scheduler, but with the ability to have users easily create the functions typically associated with traditional operating systems. In this way, it was hoped that a range of different system policies could be explored.

7.1.3. Programming Interface

Hydra provides the programmer with the basic abstraction of objects. In *Hydra*, there is a flat universe of objects (i.e., the system does not enforce any higher-level structure on objects), objects are long-lived, and they move between primary and secondary storage transparently with respect to the client. At a high level of abstraction, objects in *Hydra* adhere to the common definition of abstract data types. *Hydra* objects are typically composed of two parts — a data part and a capability part. These objects are typed, where the type of an object defines the operations that can be performed on it.

Object types are extensible in that clients can define new types (with operations), and the system provides a set of object types, examples of which are: procedures, processes, and local name spaces. The procedure-type object provides a single function, and is the basic object in *Hydra*. A *Hydra* procedure is a static entity that consists of code and a list of references to (i.e., capabilities for) the objects that are invoked from within the procedure's code. Process-type objects provide points of control in *Hydra* that can be bound to any processor in the system for execution. Each process has an associated address space that consists of the set of objects the process can access. A process's address space is defined by the capabilities that the process has in its capability part (i.e., C-list). A local name space-type object is the process's record of its current execution environment, i.e., the objects it can address within its address space. A *Hydra* procedure's local name space-type object is similar to the activation records created by language run-time systems — it is created for each invocation of a procedure-type object, and is destroyed when the invocation completes.

Hydra capabilities have access rights associated with them, indicating which operations the owner of the capability may perform on the object referenced by the capability. All objects in *Hydra* are referenced by capabilities and each object has a capability portion that contains all of the object's

capabilities. The *Hydra* system uses entities known as *templates* to describe a procedure object's actual parameters. Templates define the checks that have to be performed on the parameters. On invocation, the invoked object's template is filled in with actual parameters provided by the invoking object, and if all of the parameters pass the specified tests, the filled in template becomes an object that is placed into the C-list of the invoked object. In *Hydra*, reference counts are associated with capabilities that are used by a parallel garbage collection facility to delete objects to which no more references exist. (This task is made difficult by the fact that capabilities may be in objects that are in secondary storage.)

The scheduling of processes in *Hydra* is performed globally and is implemented in a centralized fashion. When a process of higher priority than a currently executing process is unblocked, an interprocessor interrupt is sent to a processor that is executing a lower priority task. The processor that receives such an interrupt suspends its currently executing process and dispatches the new one. In *Hydra*, long-term and short-term scheduling functions are separated to better support a range of user-defined scheduling policies. The address space of each process in *Hydra* is divided into eight parts (known as *pages*). A process may logically have an unlimited number of pages, but only eight of them are addressable at any one time. In order to accomplish this, special address relocation hardware was added to *C.mmp* (in the form of external *base registers*). Furthermore, processes in *Hydra* synchronize on data (as opposed to code) — each data structure that can be accessed by multiple processors has a lock associated with it. *Hydra* provides different synchronization mechanisms (i.e., locks and semaphores), that provided somewhat different functionality at appropriately different costs in terms of performance.

7.1.4. Analysis

Hydra was one of the earliest systems to provide support for objects and capabilities. The BLISS-11 compiler effort was performed concurrently with *Hydra* and both projects influenced each other. *Hydra* was symmetrical in that applications and system code could be executed on any processor in the *C.mmp* multiprocessor, the shared memory was made to appear uniform to the programmer, and all processors could interrupt all other processors (including themselves). Because *Hydra* was constructed for a shared-memory multiprocessor system, a number of kernel facilities could be implemented with considerably less effort than necessary in a distributed system. The programmer-transparent shared memory is used to support shared data structures. For example, garbage collection was simplified because shared memory could be used to provide common capability translation tables for all of the currently active objects, and scheduling was greatly simplified because a central-

ized mechanism could ensure that the highest priority processes in the system were always bound to the processors and there was no danger of a process getting "lost" (as they could be if processes were passed in messages).

The claim of policy/mechanism separation in *Hydra* is made for a number of facilities, but it was in fact applied only in a very restricted sense. Most typically, the facilities in *Hydra* were not devoid of policy but performed a type of parameterized policy decisions, where one of a fixed set of policies could be chosen by the client. For example, the scheduler in the *Hydra* kernel performs short-term scheduling according to a round-robin-within-priority-level discipline, that periodically returns to the higher-level for long-term scheduling decisions (which they called policies).

In the *Hydra* kernel itself, locks were used extensively for the synchronization of access to internal data structures, and therefore the cost of using the synchronization mechanisms had a great influence on the overall system performance. Furthermore, the designers proposed (but never implemented) timeouts to be associated with their semaphores to help break deadlocks.

Many of the successes and failures of *Hydra* were associated with the architecture of *C.mmp* and its processors (i.e., the symmetry of the system, uniform access times for shared memory, contention for shared pages of physical memory, and small CPU address spaces). Also, the use of asynchronous processes in *Hydra* for system functions made for simpler designs and better performance, and the symmetric nature of the system (i.e., its ability to run equally well on any processing node) made it extensible. Early attempts in *Hydra* at using different types of processors (i.e., PDP-11/20's and PDP-11/40's) were unsatisfactory because they introduced significant overhead by having the system code determine the target processor type before any code could be executed.

The system restarts that occurred when errors were detected took a quite a while to complete. Furthermore, the cost of entering and exiting the kernel caused the designers of *Hydra* to put more functionality in the kernel than might otherwise have been done. This resulted in a more complex kernel and one that exhibited a poor separation of policy and mechanism.

To enter and exit the kernel in *Hydra* took about 200 instructions (as opposed to about 50 instructions needed to enter the Alpha kernel). Furthermore, a domain change that stayed on the same processor took about 35 milliseconds (or 14,000 instructions) in *Hydra* (as opposed to 900 microseconds or 750 instructions to perform a similar function in Alpha — i.e., validate capabilities and switch to a new address space).

— Similarities

In both *Hydra* and Alpha, objects are long-lived and move between primary and secondary storage transparently with respect to the client. Processes in *Hydra* are similar to threads in Alpha, and *Hydra* procedures are much like Alpha operations. In both systems, processes can be forced to run on specific processors, but are otherwise free to run on any processor in the system. The *Hydra* call mechanism is analogous to the invocation mechanism in Alpha, and both systems use locks and semaphores to control concurrency. In both *Hydra* and Alpha, capabilities are associated with objects and used to control access among objects.

— Dissimilarities

In contrast to Alpha, which is a distributed system, the *Hydra* system was fairly centralized — it used shared memory, a shared clock, a shared processor interrupt structure, a common ready list. *Hydra* was also not intended to perform real-time command and control functions, as Alpha is, but rather to be for general-purpose, time-sharing applications.

The emphasis in *Hydra* was not on exploring reliability concepts (as is the case in Alpha), although much effort was placed on fault detection and diagnosis. High availability, and hence rapid fault recovery time, was not considered to be of great importance in *Hydra*. *Hydra* provided reliability mechanisms that could identify faults and initiate recovery procedures, and it provided the clients with an mechanism for performing explicit checkpoints of their code. Unlike Alpha, however, *Hydra* did not use atomic transaction or replication techniques to achieve higher system reliability. The *Hydra* kernel's subsystem structure allowed different facilities to compete for resources. This was appropriate given the time-sharing application domain of *Hydra*. In Alpha there is no competition among users and all resources are managed according to a uniform, global policy.

While *Hydra* made processes and procedures specific types of objects, Alpha separates the control abstraction (i.e., threads) from the definition of objects. *Hydra* provided procedures as the basic unit of object programming, whereas in Alpha the basic programming unit is an object that typically has more than one operation defined on it. The typical object in *Hydra* tended to be smaller than those found in Alpha, and the kind of objects found in Alpha would be constructed as composite entities consisting of collections of procedures mapped into a process's address space.

Hydra included a garbage collection facility that removed objects that were no longer needed (or accessible), while Alpha currently does not have such a facility. The *Hydra* system would bind

processes (transparently with respect to the client) onto processors where the required physical resources are, while the Alpha system provides all objects with physical-location-transparent access to the system's physical resources (in addition to the ability to bind objects to specific nodes).

While the objects in both systems consist of separate data and capability parts, in Alpha the data part of an object exists in a thread's virtual address space, and the capability part exists in the object's control block and therefore is not directly addressable by objects. The nature of objects in *Hydra*, and their emphasis on protection, dictated a different use of capabilities than is found in Alpha. In *Hydra*, capabilities could be used to form indirection paths to objects that do not appear directly in the invoking object's C-list. However, an Alpha object is restricted to access only those objects for which capabilities exist in its own C-list. Furthermore, *Hydra* allowed rights amplification and used capability templates to define the parameters of operations and for validating the parameters at run-time, none of which is done by Alpha.

Unlike *Hydra*, Alpha has few internal data structures that must be locked, Alpha does not use spin-locks, and internal deadlocks are not a major problem in Alpha. Also in *Hydra*, each point of control associated with an address space shares the same stack. In Alpha, separate client stacks are used to provide greater fault containment, and separate supervisor stacks are used to allow each process to be blocked independently in the kernel.

7.2. StarOS

The *StarOS* operating system was constructed at Carnegie-Mellon University by Anita Jones, et. al., from 1977 to 1982. *StarOS* was one of two operating systems built on the *CM** multiprocessor, the other being *Medusa* [Ousterhout 79]. *StarOS* is an object-oriented, capability-based operating system that was significantly influenced by *Hydra*. *StarOS* was constructed, experiments were performed on it, measurements of its performance were taken, and the system was decommissioned in 1986.

7.2.1. Hardware

StarOS was constructed on the *Cm** multiprocessor, which consisted of 50 LSI-11 microprocessors, organized in 5 clusters of 10 processors. Each processor in the system had the potential of accessing all of the memory in the system. This was accomplished through three levels of bus interconnect: starting with the Q-BUS, which was shared by a processor and its local complement of memory; followed by the intra-cluster bus, which connected the processor/memory pairs within a cluster; and finally the inter-cluster bus, which served to interconnect clusters of processing elements. There was

also a three-level hierarchy of memory access times corresponding to the three levels of interconnect — local to a processor: 3 microseconds, within a cluster: 9 microseconds, in a remote cluster: 27 microseconds. Above the Q-BUS-level of interconnect, *Cm** had highly capable, microprogrammable bus interface units, known as Slocal's (local switches) and Kmaps (cluster controllers), that were used to form, in effect, a distributed memory switch. The Kmaps supported communications concurrency and efficiency by allowing the intercluster buses to be message-switched.

7.2.2. Objectives

StarOS was designed to exploit the *Cm** hardware. The intent was to support the use of the *Cm** hardware for large, single applications — e.g., numeric-oriented tasks, and not general-purpose time-sharing. Furthermore, the *StarOS* operating system was designed to support experimentation with the notion of task forces — i.e., small processes that communicate and work together to meet a common goal.

7.2.3. Programming Interface

The programming abstractions provided by *StarOS* are similar to those provided by *Hydra* — *StarOS* supports the abstractions of task forces, objects, processes, messages, and capabilities. *StarOS* task forces correspond (dynamically) to the currently available resources in the system, and not some given (static) functionality. In *StarOS*, objects adhere to the standard definition of abstract data types, and export a set of operations that can be invoked by code in other objects. *StarOS* objects consist of two parts — a data part and capability part — both of which are contiguous regions of memory that are fixed in size. These objects have a maximum data part size of 4KB, with a maximum of 256 slots in the capability part. Processes are the units of concurrency that execute within objects and interact with other processes via message-passing. Each process has a 64KB address space, which is broken up into sixteen 4KB windows, into which objects can be mapped under the control of *StarOS*.

The invocation of operations on objects is supported by a group of system calls that make use of the system's message-passing facility. Processes use the system-provided mailbox objects to exchange data and capabilities. The invocation of operations on objects can either force the creation of a new process to perform the operation or the request could be handled by an existing server process. The parameters of invocations can consist of either data words or capabilities.

Each object in *StarOS* has a type, which defines operations on the object, and capabilities are used to reference objects and enforce access controls. Like objects, capabilities in *StarOS* have different types, and all capabilities consist of a type field, a rights field, and a data field. Capability types include: Representation, which is system-defined; Abstract, which is user-defined; Token, which identifies the owner as a possessor of special privilege; Null, which is a placeholder; and Data, which contains a single (16 bit) word. The rights associated with these capabilities include: Modify, Destroy, Copy, Restrict, C-list Write/Read/Restrict, Data Read/Write, and Type.

Clients of *StarOS* are able to define new object types, and *StarOS* provides a small set of system-defined object types. These are known as Representation Types, and instances of them are called Representation Objects. Client defined objects are known as Abstract Objects, and each has an Abstract Type along with a Representation Type (that indicates the system-defined type from which the user-defined object is constructed). The Representation Types provided by *StarOS* are: Basic Object, which is a memory segment consisting of data and capability parts; C-list, which is a Basic Object with only the capability part; Process Object, which is a schedulable entity which has a C-list associated with it; Stack Object, which provides a stack function; Queue Object, which provides a queue-oriented storage function; Directory Object, which provides a mapping function; Data Mail-box, which provides a repository for data interchange; Capability Mailbox, which provides for capability exchange; and Device Object, which encapsulates a physical device.

7.2.4. Analysis

The *StarOS* operating system is designed as a simple kernel, on top of which almost all of the standard operating system functions are implemented as task forces (i.e., objects and processes). The objects in *StarOS* tend to be fairly small and much simpler than the objects in *Hydra*.

The structure and function of the system are heavily influenced by the underlying multiprocessors' architecture. While applications make use of message-passing for their interactions, the system takes great advantage of shared memory to perform its functions. The programmer is provided with a view of a uniform, flat address space, (with different access speeds for different portions). Portions of the *StarOS* kernel were microcoded in the Kmaps, including mechanisms for low-level synchronization, memory management, message-passing, and stack and queue manipulation operations.

By migrating some of the system's functionality into microcode (in the Kmaps), the overall performance of *StarOS* is quite reasonable (e.g., a standard capability operation takes 100 microseconds, as compared to 1 millisecond in *Hydra* and roughly equivalent to the time it takes to perform a similar function in Alpha).

— Similarities

The objects (both types and instances) in *StarOS* and Alpha are similar, as well as the way in which devices are encapsulated within objects in both systems. In both systems, user-defined objects are created by invoking a create operation, indicating the type of the object to create, and the system returns to the creating object a capability with all applicable rights for the new object. The capability structure in *StarOS* and their use is similar to how capabilities are implemented and used in Alpha. Both systems make use of a language front-end. In *StarOS* there is a task force construction language, and in Alpha there is an object programming pre-processor and a configuration specification language. Furthermore, both systems are implemented as replicated kernels that run at each node, on top of which is implemented the remainder of the system functionality (along with the application) in terms of the object abstractions provided by the kernel. The manner in which *StarOS* makes use of the Kmaps to obtain communication concurrency is similar to the way Bus Interface Units are used in Alpha.

— Dissimilarities

The processes and objects in *StarOS* were intended to be smaller than those in Alpha. In *StarOS*, type managers mediate all operations on specific instance of that type of object and all access to the object's representation is done by its type manager. In Alpha, the type manager is used only to create new instances of object types, and all subsequent operations on the objects are invoked directly on the objects.

Points of control are associated with processes in *StarOS*, processes are associated with objects and they are dynamically created on invocation (if one is not already active). In contrast, threads represent control in Alpha, threads are not bound to a specific object, but move among objects as operations are invoked on them. Processes in *StarOS* can interact through asynchronous messages passed through mailbox objects. The Alpha kernel does not provide mailbox objects nor a message-passing facility, however these functions can be performed above the kernel. The *StarOS* operation invocation mechanism passes only one parameter, which is a capability for an object, that may in turn contain the capabilities and data to be used as parameters of the invocation. Also, all such parameter objects in *StarOS* contain a capability for a mailbox that is used to pass return information in and for synchronization. Operation invocations in Alpha are directly accompanied by the associated parameters, and (because of the desire to support atomic transactions) only simple RPC-like invocation semantics are used.

Unlike in Alpha, a distinction is made in *StarOS* between system- and user-provided objects (i.e., representation and abstract objects). *StarOS* uses capability paths (i.e., allows indirection through c-lists, like *Hydra*) and allows the amplification of rights, neither of which is possible in Alpha. The token capability in *StarOS* is used to provide a given manager object with complete access to one of its instances, and furthermore, *StarOS* allows representation capabilities to be sealed inside of abstract capabilities (i.e., user-defined). In Alpha, there is but one type of capability and they are significantly simpler than those in *StarOS*. In *StarOS* capabilities can be used to share a word of information without creating a shared object. It is not possible to pass raw data in capabilities in Alpha.

In *StarOS*, the cluster number where an object resides is encoded into the object's identifier, and is used for object location, which is not the case in Alpha. Objects in *StarOS* cannot not migrate freely among nodes as they can in Alpha. Furthermore, a programmer in *StarOS* must explicitly map objects into process address spaces, while the mapping of objects in Alpha is performed automatically on operation invocation.

7.3. Cronus

The *Cronus* operating system effort is being developed by Rick Schantz et.al. at Bolt, Beranek and Newman Corporation. *Cronus* is an object-oriented constituent operating system, running on a heterogeneous collection of machines connected by one or more internetworked local area networks. This work builds on the experience gained from their earlier distributed operating system work (i.e., [Forsdick 78] and [Geller 77]). *Cronus* currently runs on a collection of different machines, with differing native operating systems. Several prototype applications have been implemented and further enhancements and experimentation are currently underway.

7.3.1. Hardware

Cronus was developed on a collection of VAX processors (running UNIX and VMS), Sun Microsystems workstations (running UNIX), Masscomp workstations (bare machines), and BBN C/70's (running UNIX). All of these machines are interconnected by Ethernets within the ArpaNet internetwork environment.

7.3.2. Objectives

One of the major goals of the *Cronus* system was to achieve a high degree of interoperability among a diverse set of processing elements, and to promote and manage the sharing of resources among the processors that make up the distributed system. It was intended that *Cronus* provide a coherent and integrated system based on clusters of interconnected heterogeneous computers, in order to support distributed applications. Among the attributes desired from this effort are the survivability of system functions (i.e., reliability), the scalability of system resources (i.e., extensibility), the global management of system resources, and the convenient and efficient operation and management of the collection of processing elements that make up the *Cronus* system. Furthermore, it was indicated that an additional goal of this work is to facilitate the monitoring and control of *Cronus*, in support of the experimentation to be performed with the system. An important aspect of this work was dealing with the various system issues (frequently dismissed as "engineering details") that are necessary to make a working, usable system.

7.3.3. Programming Interface

Cronus is not a language-driven effort, but rather a constituent operating system that provides as its basic abstractions: objects, object managers, operations, and types. Each object in *Cronus* is an instance of a type, and is managed by an object management process that serves all objects of that type on a given node. All resources in *Cronus* are instances of types, and processes and directories are examples of types. Each node in *Cronus* has a (local) type manager for all of the types that exist on that node. *Cronus* supports these basic abstractions, and provides a set of system-defined object types along with their managers. Furthermore, *Cronus* provides user interface and utility routines for the client.

The *Cronus* system consists of a collection of object managers and a communications facility (known as a *communication switch*), at each node in the system. The object managers provide points of control for accessing objects of a certain type and the communication switch supports the inter-node message-passing that provides the operation invocation function among objects. *Cronus* provides location independent invocation of operations on objects, that supports dynamic relocation, replication, and extensibility. In addition to normal RPC-like operation invocation semantics, *Cronus* provides a mechanism for asynchronous message-passing that permits group (or multicast) addressing in support of object replication. The invocation of operations on objects is done in *Cronus* by sending messages (with the parameters) to the manager of the invoked object. Each object in *Cronus* has a globally unique identifier that is used in invoking operations on the object. Because the

messages generated by invocations are directed at the target object's type manager process, *Cronus* allows type managers to be referenced by using the identifier of an object it manages. To provide controls on the interaction of objects, *Cronus* makes use of a sophisticated user authentication facility and each invocation is validated through the use of an Access Control List mechanism. Access Control Lists are associated with each object and serve to indicate those users that are permitted to access the object, and which restrictions apply to each of them.

In *Cronus*, the users of the system are called principles and related users are known as groups, and both of these entities are controlled by an authentication manager. *Cronus* directory objects provide global symbolic name to object identifier mappings, and the system also maintains mappings of object identifiers to physical locations. In *Cronus*, file objects and their type managers are used to provide a distributed file system, and objects and type managers are used to encapsulate the system's physical devices. *Cronus* also provides objects for monitoring and control, and processes for the support of type managers and application programs. Furthermore, *Cronus* provides a standard library for use by clients, with conversion routines for information interchange among different types of machines in the system.

7.3.4. Analysis

Cronus executes on a cluster of machines interconnected by one or more local area networks. The majority of the processing nodes in *Cronus* are general-purpose machines, running a general-purpose timesharing operating system, along with *Cronus* as a constituent operating system. In addition to these general-purpose, utility and application nodes, *Cronus* incorporates single-user workstations, and specialized computing nodes that are dedicated to *Cronus* functions.

The object model used in *Cronus* is similar to the one in provided by the *Eden* system, and the global object directory in *Cronus* is similar to the one found in *Locus*. The object programming interface of *Cronus* is well developed and provides the programmer with a simple and highly functional interface.

The *Cronus* constituent operating system coexists with many operating systems (e.g., different versions of UNIX and VMS) and work is underway to provide native kernel support for *Cronus*. *Cronus* is written mostly in C (and a small amount of Pascal), and therefore the system is highly portable. Owing to the nature of the work, high performance is not one of the goals (or strengths) of *Cronus*.

— Similarities

As in Alpha, system (i.e., the code above the kernel) and application code are not distinguished in *Cronus*. *Cronus* uses generic object identifiers and multicast message communication in a manner similar to how logical addressing is used in Alpha. Control in *Cronus* is concentrated in the object managers and objects are passive; likewise, objects in Alpha are passive and all control is associated with threads. The interaction among type managers to manage replicated objects is similar to how the Alpha kernel manages replicated objects. Furthermore, just as in Alpha, invocations on replicated objects are transparent to the programmer and a first response selection function is used (with support for the use of different functions). In both *Cronus* and Alpha, generic, system-provided objects are used to deal with the problems of object creation (i.e., how to invoke operations on objects that do not yet exist). Also, both systems are designed to support heterogeneous machines by applying conversion functions to the contents of messages interchanged on operation invocation.

— Dissimilarities

The objectives of interoperability, heterogeneity, and internetworking make *Cronus* more of a network-like system than a logically single machine as in Alpha. Furthermore, unlike Alpha, *Cronus* is not concerned primarily with real-time command and control applications and reliability and *Cronus* (currently) does not support atomic transactions. In *Cronus*, an asynchronous message-passing facility is provided that is not available in Alpha. However, the desire for *Cronus* to function in an internetworking environment makes it impractical to use the type of logical addressing done in Alpha, and so *Cronus* must maintain logical-name-to-physical-location translation tables.

The client of the *Cronus* system is intended to be a human user, as opposed to the system software that is the intended client of the Alpha kernel. For this reason, *Cronus* supports the notion of users and groups, provides user-oriented authentication and access controls, and provides a user interface, whereas Alpha does not. Furthermore, *Cronus* is more directed towards adding on to existing operating systems, while Alpha provides a kernel on bare hardware, on which operating systems such as *Cronus* can be constructed. Also, *Cronus* only supports the exclusive form of replication, and Alpha supports both exclusive and inclusive forms of replication. The *Cronus* system uses access control lists to provide protection, while Alpha uses capabilities, and supports the construction of access list protection schemes at higher levels.

In *Cronus*, a hierarchy of programming units is defined — objects and their managers, where objects are similar to the data portion of processes and the managers provide the code typically associated

with the operations defined on objects. At compile-time, *Cronus* defines three different types of objects — primal (that do not migrate among nodes), migratory (that may migrate), and structured (e.g., replicated). On the other hand, all objects in Alpha take on any of these attributes at run-time. In *Cronus*, parameters that are passed on invocation are checked at run time and a new process is created by the object manager to handle each invocations, whereas in Alpha parameters are currently not checked at run-time and local invocations do not result in a new process being created (instead, the same thread is used and another is created only when the invocation goes to another node).

7.4. Eden

The *Eden* system was constructed at the University of Washington by Guy Almes, et. al., from 1980 to 1985. Several different instances (or prototypes) were implemented on top of different operating systems, and on different hardware. Several demonstration programs were constructed (e.g., a distributed mail system, a file system, and an appointment calendar), atomic transaction facilities were added to the system, and the work appears to be complete now.

7.4.1. Hardware

The initial *Eden* system work was done on iAPX-432-based processors, a single node prototype was later built on top of a VAX/VMS system. Work was then done on a collection of VAX processors running UNIX and interconnected via Ethernet. This work was finally extended to Sun Microsystems workstations, also running UNIX and connected by Ethernet.

7.4.2. Objectives

The *Eden* effort was directed towards the design, construction, and use of an integrated distributed system. The goals were to use an object-oriented programming paradigm, to support the construction of distributed applications on a collection of interconnected processors, and to empirically determine the programming benefits and the support costs of such a system.

7.4.3. Programming Interface

The programmer's interface to the *Eden* system is provided by what is called the Eden Programming Language, which is based on concurrent Euclid. This language provides a more natural interface to the services provided by the *Eden* system, and unlike systems such as *Argus*, *Eden* is not a language-driven effort.

The major unit of programming in *Eden* is the Eden Object, or Eject. Ejects are objects that conform to the standard definition of abstract data types (i.e., encapsulated code and data), and the size of Ejects is intended to be larger than *Smalltalk* objects, but smaller than *Argus* guardians. In many respects, Ejects are very similar to guardians in *Argus*. Ejects are active, in the sense that there are one or more processes in each Eject. The programmer's abstraction provided by *Eden* is that the processes in an Eject execute concurrently. Furthermore, as in *Argus*, processes are not only bound to particular operations provided by the Eject, but may also execute as background processes within an Eject.

Ejects (or, more accurately, the processes within Ejects) interact with other Ejects via the invocation of operations. All Ejects have globally unique identifiers that are used to exchange request and reply messages in invocations. To the programmer, invocations appear to be procedures calls, and are performed the same regardless of the physical location of the invoked Eject. Capabilities are used to access Ejects, and all invocations require that the invoker have the capability for the desired destination Eject (along with the required rights for the specified operation). The Eden Programming Language also performs type checking on the parameters that are passed in invocations.

Ejects are able to migrate among nodes in the system, and the location-transparent invocation mechanism allows operations to be invoked on them regardless of their physical location in the system. The processes in Ejects can share the memory within the Eject and the processes within an Eject interact with each other via monitors. A process that blocks on an invocation does not block the other processes in its Eject. *Eden* provides a checkpoint primitive that writes the current state of an Eject to stable storage. On a checkpoint, the system writes the information necessary to reconstruct the Eject if a crash were to occur. Ejects have a data part that contains the state of the object. This includes "long-term" state, which is maintained across invocations and "short-term" state, which consists of the parameters passed in an invocation and local variables associated with an invocation. The system supports the notion of Concrete Edentypes, which are the code segments that exist in secondary storage and define the operations (i.e., the invocation procedures) supported by instances of Ejects of this type.

Ejects can become deactivated (i.e., written out to secondary storage) either by an explicit command or on a processing node failure. Ejects may issue a command to swap themselves out in order to conserve system resources, but there is no global system policy for swapping Ejects. Deactivated Ejects are automatically reconstituted when operations are invoked on them, and they are restored to their last checkpointed state.

The *Eden* system is written in the C programming language and runs on the UNIX operating system, and consists primarily of two UNIX processes — one to support the Ejects local to the processor and the other to handle invocations. Furthermore, each Eject is implemented as a UNIX process, and the processes inside the Eject use the UNIX IPC facility to perform invocations. Ejects can make use of the facilities of the underlying UNIX system (e.g., file system or I/O).

7.4.4. Analysis

The *Eden* system relies on the Eden Programming Language to provide fault containment. Atomic transactions are not directly supported in the *Eden* system, but *Eden* does provide a checkpoint mechanism. However, the checkpoint operation was difficult to implement because of conflicting policies of the underlying UNIX operating system. Additionally, the performance of *Eden* suffered greatly from the fact that it attempted to implement operating system functionality on top of an existing operating system with different functionality (i.e., UNIX). By implementing system entities that are not visible to the underlying system (i.e., processes within Ejects), these resources cannot be managed by the system effectively. For example, should a process in an Eject take a page fault, the whole UNIX process (including all of the other processes in the Eject) is blocked. Because of the poor performance and general difficulty of implementing an operating system on top of another system with different (and possibly incompatible) goals, it was difficult to construct practical, distributed applications on *Eden*.

In *Eden*, capabilities are not strongly typed and the rights associated with capabilities are not system-defined (as they are in *Hydra*), but rather are object-defined. Also, the asynchronous invocation mechanism is not used extensively in *Eden*, bringing into question the usefulness of such a service. Initial performance measurements of *Eden* implemented on a VAX processor indicated that local, synchronous invocations were over 100 milliseconds, while a more recent implementation of *Eden* on Sun Microsystems workstations has reduced this figure to 50 milliseconds for a null invocation and return (this compares to less than 1ms for similar functions in Alpha). The underlying UNIX operating system made atomic updates of files difficult in *Eden* and consequently checkpoints of Ejects take about one second to complete. Each of the processes that constitute the *Eden* run-time system consists of approximately 80KB of code.

— Similarities

The language interface to *Eden* is similar in intent and usage to that of Alpha. In both cases, the language generates the necessary stubs for marshaling and unpacking parameters at the source and

destination ends of an invocation. Both systems provide RPC-like semantics for invocations and achieve concurrency by having multiple points of control active in an Eject/object. Alpha and *Eden* use capabilities in a similar fashion, to control interactions among objects. The update operation provided for objects in Alpha is analogous to the checkpoint operation provided in *Eden*; both operations save only the "long term" state of the objects in non-volatile storage. As in Alpha, *Eden* does not provide the user with a file system, in the traditional sense, but rather the definitions of objects/Ejects subsume the functionality typically associated with file systems. Furthermore, as in Alpha, *Eden* does not attempt to provide the user with a reliable virtual machine, but rather a collection of mechanisms with which to build reliable applications. Also, both systems indicate the desire to provide some form of garbage collection.

— Dissimilarities

In *Eden* there is a two-layer hierarchy of programming units — Ejects and processes — whereas Alpha presents the programmer with a flat universe of objects. Also, processes in Ejects interact through monitors, while the multiple points of control that may be active in an object in Alpha may make use of mechanisms that permit a higher level of concurrency than available through monitors. In both systems there are a number of communication mechanisms that provide different levels of semantics, however in *Eden* these services are not unified into a common invocation mechanism as they are in Alpha.

In *Eden*, an Eject must be checkpointed before it can become recoverable, while objects in Alpha can be made recoverable when they are created or at run-time. Also, in *Eden* explicit actions are required on the part of Ejects in order to swap out objects, while in Alpha objects that have not been accessed in some period of time are automatically swapped out. In the *Eden* system it is necessary to maintain caches of mappings between capabilities (i.e., the logical references to Ejects) and the physical machines where they reside. This is not necessary in Alpha because of the high-level logical addressing structure is carried down to the communication subnetwork. In *Eden*, all of an Eject's state is written out on a checkpoint, whereas in Alpha logging is used to write out only the changes to an object when it is checkpointed. Furthermore, the notions of atomic update and permanence of storage are not separated in *Eden* as they are in Alpha. Capabilities are implemented differently in Alpha and *Eden*. Because of the perceived high cost of entering the kernel to examine capabilities, *Eden* maintains a copy of each Eject's capability list within the Eject. In this manner, Ejects can access this "capability cache" at high speed, while the kernel maintained capability list is still the ultimate authority.

In *Eden*, points of control are associated with processes that are bound to specific Ejects. While in Alpha, points of control are associated with threads that begin in an object and move among other objects. While these approaches are not significantly different when threads cross node boundaries, the thread approach in Alpha offers a significant cost savings in terms of performance due to the need to do only a partial context switch, and the reduction in the number of control structures required (to manage a smaller number of processes).

7.5. Argus

Argus is a language (and associated run-time support system) for the construction of distributed programs, being developed by Barbara Liskov, et.al. at MIT since 1979. The initial work is on a centralized system, running on top of UNIX. Work is underway, however, to move to a decentralized system and to implement a native run-time system. Great emphasis has been placed on the reliability aspects of constructing distributed programs, resulting in significant work in the area of atomic transactions and replication.

7.5.1. Hardware

Argus has been implemented on a VAX processor (running UNIX). Work is underway to extend this to several VAX processors interconnected by a local area network.

7.5.2. Objectives

Argus attempts to provide an integrated language (and run-time support system) to permit the construction of distributed programs. In the *Argus* programming model, distributed programs are to run on a collection of processing nodes interconnected by some communication network. The potential benefits to be explored in this system include: increased availability of service, enhanced extensibility of the system, autonomy of control for the processing elements, physical distribution of the system's processing elements, increased concurrency of operations, and consistency of the information maintained by the system.

7.5.3. Programming Interface

The *Argus* programming interface is provided primarily by an dialect of the CLU programming language. The main programming abstractions provided in *Argus* are guardians and handlers. In *Argus*, distributed programs are created from a collection of communicating guardians. Guardians can be considered as virtual nodes interconnected by a communication network, with stable data storage and unlimited amounts of primary memory. Like objects, guardians encapsulate data and contain handlers that are procedures that provide the only means of accessing the data contained within the guardian. Each guardian may contain a number of processes, that execute in asynchronous concurrency with respect to all other processes in the system. In *Argus*, processes are light-weight points of control that share access to the data within a guardian. Each operation defined on a guardian (i.e., each handler) has associated with it a separate process within the guardian. The *Argus* system supports the notion of atomic actions to synchronize concurrent access to the shared data within guardians.

In *Argus*, a processing node is defined to consist of one or more processors, with one or more levels of memory, and any local peripheral devices. The inter-node interconnection can be a local-area or wide-area network, but not shared memory. Furthermore, the *Argus* application domain is one where the preservation of long-lived data is very important (airline reservations, office automation, databases, etc.), and response time is not as important. The CLU programming language is strongly typed, supports data and control abstraction, and includes automatic garbage collection.

In *Argus*, CLU has been extended to include support for distribution, concurrency control, and atomic actions. The invocation of handlers in guardians is provided by the language in a location-transparent fashion; processes invoke operations on guardians identically regardless of the physical location of the destination guardian.

Argus also provides the means of creating critical regions to synchronize the processes within a guardian, and it supports atomic actions with commit and abort primitives, along with stable storage. Furthermore, in addition to the processes that are bound to operations/handlers in a guardian, there can be additional processes in a guardian that provide independent activity (that can be used to perform housekeeping or background functions within a guardian).

Currently, *Argus* runs on UNIX; each guardian is implemented as a UNIX process, and invocations are performed by an RPC service constructed on top of the UNIX IPC facility. The language run-time support system provides *Argus* processes (i.e., those inside guardians), and guardian storage manage-

ment. Each *Argus* (internal) process has its own stack segment, transient data segment, and process control block, and each guardian is provided with its own scheduler for the processes inside. The *Argus* run-time system provides a process scheduler that does preemptive, round-robin scheduling, and there are two different types of interrupts: a periodic clock interrupt used for time-slicing, and an interrupt that indicates a message arrival for this guardian (i.e., UNIX process). Furthermore, stable storage in *Argus* is simulated through the use of the UNIX raw disk interface (i.e., disk block read and write commands). However, UNIX does not adequately support the programming abstractions of *Argus*, nor does it provide sufficient support for high-bandwidth communication or garbage collection of a large virtual address space. Interest, therefore, has been expressed in providing kernel support for *Argus* [Allen 85].

7.5.4. Analysis

Argus is primarily a language effort that is concerned with issues of constructing reliable distributed applications. However, *Argus* is not an operating system effort, despite the fact that many of their programming abstractions call for operating system support. Furthermore, *Argus* introduces two layers of programming entities, where the lowest entities are not visible to the underlying system, i.e., UNIX only supports the processes that are used to implement guardians and it has no direct knowledge of the *Argus* processes within guardians. This causes all processes within a guardian to block should any process generate a page fault. In *Argus*, interrupt handling is deferred until procedure calls have completed, thereby simplifying synchronization problems. Also, while full UNIX context swaps are required when a process in another guardian is to run, the context swap required among processes in the same guardian is very fast (i.e., 13 instructions).

— Similarities

The synchronization in *Argus* is similar to that provided by Alpha, the *Argus* "mutex" data type is similar to the Alpha lock mechanism. Both of these mechanisms are used to ensure that the associated data is only written to stable storage when it is in a consistent state. In *Argus* user-defined, nested transactions are implemented in a fashion similar to how they are performed in Alpha, and a similar set of restrictions are applied to ease the implementation difficulties (e.g., RPC semantics and simple nesting). Furthermore, *Argus* and Alpha use similar two-phase locking and two-phase commit schemes for their transactions. Both of these systems use timeouts and aborts on transactions to break deadlocks, and use replication to enhance availability of services.

— Dissimilarities

In contrast to the objectives of the Alpha kernel, the individual processing elements in *Argus* comprise a collection of autonomous systems in an internetwork environment, as opposed to a logically singular system. In the Alpha kernel, the emphasis is on providing kernel-level mechanisms, while the emphasis in Argus is on providing programming language primitives. The *Argus* system supports a two-level hierarchy (with guardians and processes), while Alpha provides a flat universe of objects. In *Argus* messages are copied between the UNIX kernel and processes in guardians, whereas the Alpha kernel uses page mapping. The processes within guardians do not exist in separate addressing domains, and therefore fault containment is provided among processes executing within guardians, in the manner in which threads executing within objects in Alpha are protected. *Argus* provides atomic actions by way of a single (monolithic) atomic transaction primitive, while transactions are provided in Alpha by a collection of independent mechanisms. Furthermore, in *Argus* a guardian is declared to be atomic at compile time, whereas in Alpha clients may dynamically choose (at run-time) the characteristics objects are to take on.

7.6. Accent

The *Accent* kernel was developed by Rick Rashid, et. al., at Carnegie-Mellon University in 1981. *Accent* was designed as the basis for a large scale network of high-performance personal workstations. *Accent* is a kernel that provides process support and an IPC facility, and all of the subsequent layers are constructed out of processes that use the message-passing primitives of the kernel. A program known as Matchmaker takes high-level language specifications of RPC-like invocations, and generates the appropriate stubs for the source and destination processes. *Accent* has been in use since 1981, and is running on over 200 workstations at Carnegie-Mellon alone. *Accent* has been ported to a number of workstations and mainframes, and is marketed and sold internationally.

7.6.1. Hardware

Accent was originally developed for the Perq workstation, a single-user workstations, based on a custom bit-slice processor (with a 170nsec microcycle time), with 20-bit physical addresses (with no memory management hardware), and 1MB of primary memory (with 340nsec access time). The Perq originally had a 4K word microstore (later upgraded to 16K words) that Accent used to implement microcoded support for virtual memory and interprocessor communication. Furthermore, the Perq workstations could be interconnected via an Ethernet.

7.6.2. Objectives

Accent was designed as a kernel for the SPICE project at CMU, in order to provide the basis of an operating system for a collection of high-performance workstations in a computer science research environment. Among the major goals of this effort were extensibility, protection, and network-transparency. The system was to provide a simple, uniform interface supporting processes and based on a high-performance IPC facility through which all system- and user-provided services can be accessed. The kernel was to support multiple processes with large, sparse paged address spaces, and provide the ability to pass large amounts of (possibly structured) data in messages. Furthermore, the kernel was to be run on a range of (heterogeneous) hardware bases, ranging from mainframes to individual workstations.

7.6.3. Programming Interface

The *Accent* kernel provides support for processes and a protected, message-based IPC facility. In Accent, the basic abstraction is the process and all interactions are performed via the IPC facility. All system services are accessible via the IPC facility, thus providing uniform access to all system- and user-defined resources. The *Accent* IPC facility provides the message and port abstractions. Messages consist of a fixed-size header followed by a variable sized (up to 2^{32} bytes), self-describing collection of data items. Ports are protected kernel entities to which messages can be sent and from which they can be received. Ports are manipulated via capabilities, that can be passed among processes in messages. In *Accent* a decision was made to define all message interfaces in a high-level specification language. Matchmaker is a language interface that provides an object-invocation type of interface on top of *Accent*. Matchmaker allows both synchronous and asynchronous calls to be made, and generates stubs that handle the packing and unpacking parameters, and waiting for replies (if necessary), for both the sources and destinations of logical calls. Matchmaker currently works with the C, Pascal, ADA, and Common Lisp programming languages.

Accent provides the kind of system that is physically distributed, but loosely coupled logically. *Accent* supports a notion of multiple users, each having control over their own, autonomous machines, that may cooperate with other users' machines. Because the hardware on which *Accent* was first implemented (i.e., the Perq) had no support for virtual memory management, it was supported in microcode. The Accent virtual memory facility supports large, separate virtual address spaces for each process, provides the ability to map pages from files into a process's address space, and uses copy-on-write page mapping to increase message-passing performance.

7.6.4. Analysis

Accent is one of the few real operating systems constructed in academia in the last few years, and represents a data point in a region of the kernel design space that had previously not been explored. It provides a kernel based entirely on processes and messages, and a multitasking, virtual memory-based system for a collection of powerful, single-user workstations. As such, *Accent* provides some of the most current design and implementation experience for this class of systems.

Measured across time, it has been determined that approximately three-fourths of the calls generated by Matchmaker are synchronous, RPC-like invocations, and most of the asynchronous calls are attributable to the ADA compiler. On a Perq, a typical Pascal procedure call takes around 60 microseconds, while a simple local message-send takes around 1.25 milliseconds, and a remote message-send takes around 2.5 milliseconds.

— Similarities

The Alpha and *Accent* kernels are constructed on similar hardware bases, i.e., a collection of high-performance workstations, interconnected by a local area network. The *Accent* IPC facility is analogous to the operation invocation facility in Alpha, just as *Accent* processes can be considered analogs of threads and objects in Alpha. Both systems are based on the support of a basic unit of concurrency and programming, along with a global communication facility, and the systems derived similar benefits from this simple and uniform programming interface. *Accent* and Alpha both use a capability mechanism for controlling communications (and hence interaction among the units of programming). The virtual memory facility provided by *Accent* is similar to that provided by Alpha, and in both cases the majority of the kernel can be paged (unlike systems such as UNIX that have large kernels which are locked in memory). Another similarity between the two systems occurs in the way in which virtual memory mechanisms are used to enforce the separation of address domains for each programming unit, to enhance the performance of context switches among units, and to provide increased performance in inter-domain information exchanges (through mapping shared pages between domains and such techniques as copy-on-write). In both systems, considerable effort has been put into optimizing (synchronous) communications among programming units both within and across processing nodes. Furthermore, the size granularity of processes in *Accent* is similar to that of objects in Alpha.

— Dissimilarities

While *Accent* uses Matchmaker to create an object-like interface, the Alpha kernel provides an object-oriented programming interface directly. Furthermore, *Accent* (unlike Alpha) does not place reliability among its major system objectives, and it does not attempt to logically integrate the processors that make up the distributed system. In Alpha, hardware resources are used somewhat more creatively than in *Accent*. In *Accent* the data passed in messages can be considerably larger than that which can be passed as parameters to operation invocations in Alpha.

7.7. Locus

The *Locus* system was developed by Gerald Popek at UCLA, beginning in 1980. This system began as an attempt to extend the UNIX file system across multiple machines and ended up becoming a distributed UNIX system. The system is in use at UCLA and is being sold commercially. In recent times, various features, such as locking and atomic transactions, have been added to *Locus*, and work is continuing on the system.

7.7.1. Hardware

Locus was developed (and currently runs) on a collection of VAX mainframes and Motorola MC68000-based workstations.

7.7.2. Objectives

The main goal of *Locus* is to make a collection of computers (either individual workstations or communal mainframes) as simple to use as a single, uniprocessor system. The application domain for *Locus* was chosen to be general-purpose timesharing, office automation, or database applications. *Locus* attempts to address the issues of application-transparent extensibility, processor heterogeneity, reliability and availability. To achieve these goals it was deemed necessary that *Locus* provide a network-transparent, reliable, distributed file system. Furthermore, this was to be done without increasing the cost of local file access and minimizing the difference in cost between local and remote file accesses.

7.7.3. Programming Interface

The programming interface provided by *Locus* is that of a user-transparent distributed UNIX system. The user is transparently provided with a reliable, distributed file system, that automatically supports the replication of files. The standard UNIX interface has been extended to include a nested atomic transaction facility. This facility is supported by client-level BeginTransaction, EndTransaction, and AbortTransaction primitives. Furthermore, a locking primitive exists that permits regions of files to be locked, in various different modes. The transaction facility is designed to accommodate the sharing of files among processes both within and outside of transactions.

Fundamentally, *Locus* is distributed UNIX, with atomic transactions and file locking. The *Locus* file system ensures that files are kept consistent even in the face of machine failures (i.e., either all of the updates to a file are made or the file remains unchanged). All of the distribution and reliability features in *Locus* are compatible with UNIX, and are supported primarily through the file system. In *Locus*, the transaction and replication facilities exhibit a high degree of performance (as per the system objectives). Locking is done on regions of files, and a two-phase locking discipline is used to ensure serializability of transactions. To ensure that the transaction guarantees can be met, even when sharing files with processes that are not in transactions, it is required that processes always lock the files they access.

7.7.4. Analysis

Locus does not provide an object-oriented programming interface. More than anything else, *Locus* is a distributed file system effort, performed in the context of UNIX. The functionality of *Locus* — distribution and reliability — is primarily provided through modifications and additions to the UNIX file system. Furthermore, the functionality provided by *Locus* is heavily influenced by its UNIX context. The result was a number of positive effects including the fact that the system was developed from a stable pre-existing system base, the system has a large software base is available to it, and the system is able to meet the objectives of user-transparent distribution and simplicity of the user's interface. Also, much of the efficiency in the *Locus* facilities is attributable to the fact that these functions are associated with the file system. However, this file-system-oriented approach tends towards larger granularities in application programming solutions. Interactions among processes is primarily based on the use of shared files, and the entities that are locked by processes within transactions are regions of files. While the *Locus* features may not contribute much to the cost of normal file operations, typical UNIX file manipulation operations tend to be costly to practically permit the fine-grained types of interactions that might be associated with data items within objects.

— Similarities

The policy applied to the use of locks in *Locus*, by processes both within and outside of transactions, is similar to that of the Alpha kernel. In both cases the systems ensure that all locked data items are committed (i.e., written to storage), even if they were accessed by a process that is not in a transaction, and even in cases of read-only accesses. The transaction commit and abort procedures are also very similar to those used in the Alpha kernel. Furthermore, both systems use similar techniques for doing careful disk writes to ensure the atomic update of stored data.

— Dissimilarities

Locus provides a UNIX interface, which differs from the object-oriented interface of Alpha in all of the obvious ways that process- and object-oriented systems differ. The interesting dissimilarities between *Locus* and Alpha have to do with the reliability features provided by these systems. The transactions and replication facilities in *Locus* deal with parts or all of files, while in Alpha the data items associated with these facilities are parts or all of objects. In *Locus*, locked sections of files have shadow pages associated with them to permit atomic update of files on commit and roll-back on abort or transactions. This is practical for *Locus* because their locked data items are based on files. In Alpha, the data items locked can be arbitrary portions of objects and so write-ahead logs are used for smaller data items, and shadow page techniques are used for larger regions of data. Furthermore, all of the functions provided by *Locus* are implemented in software, with no special hardware support, while the Alpha kernel takes advantage of the addition of hardware support for operating system functions.

- UNIX is a trademark of AT&T Bell Laboratories.
- VAX, PDP-11/40, PDP-11/20, LSI-11, Q-BUS, UNIBUS, BLISS-11, and VMS are trademarks of Digital Equipment Corporation.
- iAPX-432 is a trademark of Intel Corporation.
- Perq and Accent are trademarks of Perq Corporation.
- Locus is a trademark of the Locus Corporation.
- Ada is a trademark of the United States Department of Defense.
- Ethernet is a trademark of Xerox Corporation.

Appendix A

Programming Language Extensions

The work described here does not include an actual compiler effort. It was, however, necessary to provide some rudimentary language support for the programming interface supported by the Alpha kernel. This section describes the language constructs provided by the simple pre-processor used to generate objects for the kernel. This section also includes a simple example of the use of the various language primitives described here.

These simple extensions are not an attempt to provide a well-integrated language interface for the client. The pre-processor provides nothing more than a simplified interface to the mechanisms provided by the kernel. In particular, there is an obvious need for higher-level synchronization constructs, block structured constructs for transactions and deadlines, and some form of exception handling construct (especially for invocation status returns).

A.1. Keywords

The primitive extensions to the C programming language that provide support for the abstractions in the Alpha kernel take the form of a set of new keywords. The additional keywords and their usage are defined as follows:

A.1.1. Object Declaration

OBJECT: This keyword is used to declare an object, provide it with a symbolic name, and an indication of the attributes, optimizations, or restrictions that the object should take on. Instances of the specified type of object assume the default attributes for each qualifier that is not explicitly specified.

ATOMIC-UPDATE:
This qualifier is used in object declarations to indicate that the secondary storage image of the object being declared should be updated atomically with respect to external visibility and system failures. If this qualifier is not used, the default case

applies, i.e., no attempt is made to have updates to the object's secondary storage image be atomic.

KERNEL: This keyword is used in object declarations to indicate that the object being declared should be implemented as a kernel object. If this qualifier is not used, instances of this type of object are created as client objects.

PERMANENT: This keyword is used in object declarations to indicate that the secondary storage image of the object being declared should be maintained in non-volatile storage and reconstituted after node failures. If this qualifier is not used, the default case stores the object's secondary storage image in a volatile portion of secondary storage.

RESTRICTIONS:
This keyword is used in object declarations to restrict the rights associated with the capability passed to the creator of an instance of the type of object being declared. This keyword is followed by a list of parameters that indicate the restrictions that are to be applied to various operations of the object.

EXCLUSIVE-USE:
This keyword is used in conjunction with the RESTRICT and RESTRICTIONS keywords to indicate that only the thread that obtained the capability can make use of it.

NO-COPY: This keyword is used in conjunction with the RESTRICT and RESTRICTIONS keywords to indicate that the capability for the given operation cannot be copied — i.e., if the capability is passed in an invocation, it is removed from the invoking object's c-list.

NO-MULTIPLE-USE:
This keyword is used in conjunction with the RESTRICT and RESTRICTIONS keywords to indicate that the capability for the given operation can be used only once.

NO-TRANSFER: This keyword is used in conjunction with the RESTRICT and RESTRICTIONS keywords to indicate that the capability for the given operation cannot be passed to another object as a parameter of an operation invocation.

A.1.2. Operation Declaration

OPERATION: This keyword is used to declare an operation within an object. This keyword provides a logical name for the an entry point into an object, along with the operation's formal parameter list.

IN: This keyword is used in the formal parameter lists of operation declarations to indicate that the following parameters are to be passed to the operation from the invoking object. Following this keyword are a list of one or parameter specifications (i.e., type and variable-name pairs).

OUT: This keyword is used in the formal parameter lists of operation declarations to indicate that the following parameters are to be passed from the invoked object when the operation completes. Following this keyword are a list of one or parameter specifications.

A.1.3. Capability Declaration

CAPA: This keyword is used like a data type specification in declaring capabilities. This keyword is followed by the name of the type of object that the capability is to refer to and a list of the identifiers for the individual capabilities being declared.

WELLKNOWN: This keyword is used in conjunction with the CAPA keyword to indicate that the specified capability is to be provided to the object by the kernel when an object of this type is created.

A.1.4. Operation Invocation

INVOKE: This keyword is used for the invocation of operations on objects and is followed by a capability for the destination object, the name of the operation to be invoked, and the invocation's actual parameter list.

RESTRICT: This keyword is used to apply restrictions to capabilities being passed as parameters of operation invocations. This keyword is followed by the capability to be restricted and a list of the restrictions to be applied to it.

RETURN: This keyword is used to indicate the completion of operations, in place of the normal C **return** keyword. This keyword is followed by an indication of the success or failure of the operation.

SUCCESS: This keyword is used following the **RETURN** keyword to indicate that the invoked operation completed successfully.

FAILURE: This keyword is used following the **RETURN** keyword to indicate that the invoked operation failed. This keyword may be followed by an (optional) indication of the reason for the invocation's failure.

A.1.5. Thread Deadline Specification

DEADLINE: This keyword is used to define a region within an operation that has a time constraint associated with it. This keyword is followed by a collection of parameters that indicate the attributes of this deadline block and the region of an object's code that makes up the block. The current deadline parameters are: an indication of the amount of (elapsed) time this block of code should take to complete, an estimate of the amount of computation time required by this block, and an indication of the type of deadline this is to be (currently, *hard* or *soft*).

HARD: This keyword is used with the DEADLINE keyword to indicate that there is no value in continuing the specified computation after it has been determined that its deadline cannot be met.

SOFT: This keyword is used with the DEADLINE primitive to indicate that there is some value to continuing the specified computation after it has been determined that its deadline cannot be met.

A.1.6. Miscellaneous Primitives

NODE: This keyword has the value of the node identifier of the node at which the invoking thread is currently executing.

THREAD: This keyword has the value of a unique identifier for the currently executing thread.

RESULT: This keyword has a value which reflects the result of the last operation invoked by the currently executing thread. This keyword has value 0 if the operation returned SUCCESS, -1 for FAILURE returns without a specified reason, or the value specified with the FAILURE qualifier to the RETURN keyword.

A.2. Syntax

The following is a description of the syntax for object programming language extensions. In this description, the added keywords appear here in boldface, while parts with standard C syntax are represented in italics.

A.2.1. Object Declaration

object-header ::= **OBJECT** '(' object-identifier object-spec ')'

object-spec ::= { ':' object-attribute-list }? { ';' **RESTRICTIONS** ':' restriction-spec-list }?

object-identifier ::= *C-identifier*

object-attribute-list ::= object-attribute { ',' object-attribute }*

object-attribute ::= **ATOMIC-UPDATE | KERNEL | PERMANENT**

restriction-spec-list ::= restriction-spec { ',' restriction-spec }*

restriction-spec ::= '(' operation-identifier ':' restriction-list ')'

restriction-list ::= restriction { ',' restriction }*

restriction ::= EXCLUSIVE-USE | NO-COPY | NO-MULTIPLE-USE | NO-TRANSFER

A.2.2. Operation Declaration

operation-header ::= OPERATION operation-identifier '(' { parameter-spec-list }? ')'

operation-identifier ::= *C-identifier*

parameter-spec-list ::= parameter-spec { ',' parameter-spec }*

parameter-spec ::= { IN | OUT } parameter { ',' parameter }* ';'

parameter ::= *C-type-spec* ':' *C-identifier*

A.2.3. Capability Declaration

capa-declaration ::= { WELLKNOWN }? CAPA capa-spec ';'

capa-spec ::= object-identifier identifier-list

capa-list ::= *C-identifier* { ',' *C-identifier* }*

A.2.4. Operation Invocation

operation-invocation ::= INVOKE target-spec '(' { invocation-parameter-list }? ')' ';'

target-spec ::= object-identifier '.' operation-identifier

invocation-parameter-list ::= *C-identifier* { ',' *C-identifier* }*

operation-return ::= RETURN return-spec ';'

return-spec ::= SUCCESS | FAILURE { '(' failure-cause ')' }?

failure-cause ::= *C-expression*

A.2.5. Thread Deadline Specification

deadline-block ::= **DEADLINE** '(' deadline-time ',' expected-computation-time ',' deadline-type ')'

deadline-time ::= *C-expression*

expected-computation-time ::= *C-expression*

deadline-type ::= **HARD** | **SOFT**

A.2.6. Miscellaneous Primitives

current-node ::= **NODE**

current-thread ::= **THREAD**

invocation-result ::= **RESULT**

A.3. Example

The following example illustrates all the additional keywords. The presentation conventions for this example are:

- **The standard C language keywords appear like this.**
- **THE ADDITIONAL KEYWORDS APPEAR LIKE THIS.**
- Standard C operators, procedure names, variables, user-defined types, object names, and operation names all appear like this.
- *Comments appear like this.*

This is the obligatory banking example, which seems to accompany all discussions of atomic transactions. In this example there are four objects — a console interface object (***Console***), a control object (***Control***), an account management object (***Account***), and an audit trail object (***Audit***).

This application program provides a user at a console device with a command interface, which supports the manipulation of a canonical banking application. The operations that the user can perform are: **deposit, withdraw, balance-query, and transfer.**

The *Control* object is the point where the threads in this example begin execution. Following the initialization of the other objects, the *Control* object invokes an operation on the *Console* object to obtain a command from the user. The *Console* object is responsible for the user-interface functions. It provides user prompting, input validity checking, and command line editing. When a user issues a valid command, the invocation on the *Console* object returns and the *Control* object then invokes the desired operation(s) on the *Account* object. The operation(s) invoked by the *Control* object on behalf of the user are done within an atomic transaction, and once the specified operations have been successfully performed, the *Control* object commits the atomic transaction and begins the cycle again. The *Account* object performs the desired operations invoked upon it by the *Console* object and invoke an operation on the *Audit* object to log the operations performed on the account database. Also, these operations are performed within an atomic transaction.

This example assumes that the objects can be arbitrarily distributed among the system's physical nodes, and that there is a higher-level object that creates threads and provides them with their initial parameters. It is furthermore assumed that other objects are responsible for creating the threads which begin in the *Control* object, and for initializing the *Account* object.

The *Account* and *Audit* objects are permanent and atomically updateable. These objects survive node failures and they cannot be seen in an inconsistent state. The *Console* and *Control* objects do not retain any important state information and therefore are transient objects. Additionally, the *Console* object is a kernel object that encapsulates a physical device.

For the sake of brevity, some of the lower-level subroutines are omitted.

```
/*****************************************************************************\
*                                                                             *
*                                  CONSOLE.H                                  *
*                                                                             *
\*****************************************************************************/


/* types of commands */
typedef enum cmd {
        DEPOSIT,
        WITHDRAW,
        TRANSFER,
        QUERY,
        INVALID
} cmd_t;

/* type of parameters that accompany commands */
typedef struct parmblk {
        int     srcid;
        int     dstid;
        int     amt;
} parmblk_t;
```

```
/*****************************************************************************\
*                                                                             *
*                                  CONSOLE.C                                  *
*                                                                             *
\*****************************************************************************/


/*
 * Object Declaration
 */

OBJECT(Console: KERNEL)


/*
 * Include Files
 */

#include        "console.h"


/*
* Declarations
*/

#define BUFSIZE         80      /* size of text input buffer */
```

```
/*
 * Operations
 */

OPERATION  Get_Command(OUT cmd_t:cmd, parmblk_t:parmblk)
/*
 * This operation prompts for input from the user, performs the necessary
 * command line editing functions, and validates the given command.
 * It also is responsible for obtaining the parameters associated with
 * each operation.  It returns a parameter which indicates the type of
 * command which was issued, and a structure which contains the parameters
 * associated with the given command.
 */
{
    char        str[BUFSIZE];

    /* get a valid command */
    do {
        /* prompt for new command and get an input line from the user */
        Message("Command? ");
        GetString(str);

        /* determine if the command is valid */
        cmd = Validate(str);
    } while (cmd != INVALID);

    /* get the parameters for the command */
    switch (command) {
        DEPOSIT:
        WITHDRAW:
            /* get an account id */
            Message("Account? ");
            parmblk.srcid = GetAccountID();

            /* get an amount */
            Message("Amount? ");
            parmblk.amt = GetValue();
            break;
        TRANSFER: {
            /* get an account id's */
            Message("From Account? ");
            parmblk.srcid = GetAccountID();
            Message("To Account? ");
            parmblk.dstid = GetAccountID();

            /* get an amount */
            Message("Amount? ");
            parmblk.amt = GetValue();
            break;
        };
        QUERY: {
            /* get an account id */
            Message("Account? ");
            parmblk.srcid = GetAccountID();
            break;
        };
    };

    /* return a success indication */
    RETURN SUCCESS;
};
```

```
OPERATION  Fail()
/*
 * Print a failure message on the console.
 */
{
    /* print negative ack */
    Message("Command Failed");

    /* return success */
    RETURN SUCCESS;
};


OPERATION  QueryResponse(IN int:acct, int:amt)
/*
 * Print a message on the console in response to an account query.
 */
{
    /* print the message */
    Message("Account: %d, Balance: %d", acct, amt);

    /* return success */
    RETURN SUCCESS;
};
```

```
/*****************************************************************************\
*                                                                             *
*                                  CONTROL.C                                  *
*                                                                             *
\*****************************************************************************/

/*
 * Object Declaration
 */

OBJECT(Control)


/*
 * Include Files
 */

#include        "console.h"


/*
 * Declarations
 */

WELLKNOWN CAPA Account            Acct;
WELLKNOWN CAPA ObjectManager      ObjMngr;
WELLKNOWN CAPA TransactionManager TransMngr;

#define SUCCESSFUL    0
```

```
OPERATION  Start()
/*
 * This is the main entry point of the control. This operation coordinates the
 * operations of the banking application. A thread begins in this operation for
 * each client that is currently connected to the system by a console.
 */
{
    int                 amount;
    cmd_t               cmd;
    parmblk_t           parmblk;
    CAPA Console        Cons;

    /* create a new instance of a console object */
    INVOKE ObjMngr.Create(Console, Cons);
    if (RESULT != SUCCESSFUL) Error();

    /* the main command/action loop */
    while (1) {
        /* get a command */
        INVOKE Cons.GetCommand(cmd, parmblk);
        if (RESULT != SUCCESSFUL) Error();

        /* start a transaction */
        INVOKE TransMngr.Begin(BASIC);
        if (RESULT != SUCCESSFUL) Error();

        /* carry out the given command */
        switch (cmd) {
            case DEPOSIT: {
                INVOKE Acct.Deposit(parmblk.srcid, parmblk.amt);
                break;
            };
            case WITHDRAW: {
                INVOKE Acct.Withdraw(parmblk.acctid, parmblk.amt);
                break;
            };
            case TRANSFER: {
                INVOKE Acct.Withdraw(parmblk.srcid, parmblk.amt);
                if (RESULT == SUCCESSFUL)
                    INVOKE Acct.Deposit(parmblk.dstid, parmblk.amt);
                break;
            };
            case QUERY: {
                INVOKE Acct.Query(parmblk.srcid, amount);
                if (RESULT == SUCCESSFUL)
                    INVOKE Cons.QueryResponse(parmblk.srcid, amount);
                break;
            };
        };
        /* check the result of the issued command(s) */
        if (RESULT != SUCCESSFUL) {
            /* notify the user of the failure */
            INVOKE Cons.Fail();
            if (RESULT != SUCCESSFUL) Error();

            /* abort the transaction */
            INVOKE TransMngr.Abort()
            if (RESULT != SUCCESSFUL) Error();
        };

        /* end the transaction */
        INVOKE TransMngr.End();
        if (RESULT != SUCCESSFUL) Error();
    };
};
```

```
/*****************************************************************************\
*                                                                             *
*                                  ACCOUNT.C                                  *
*                                                                             *
\*****************************************************************************/

/*
 * Object Declaration
 */

OBJECT(Account: PERMANENT, ATOMIC-UPDATE)


/*
 * Declarations
 */

#define SUCCESSFUL      0
#define MAXACCTS        100;

int     accounts[MAXACCTS];

WELLKNOWN CAPA SemaphoreManager  SemMngr;
WELLKNOWN CAPA ObjectManager     ObjMngr;
CAPA   Sem;
CAPA   Audt;


/*
 * Operations
 */

OPERATION  Init()
/*
 * Initialize the database.
 */
{
    int i;

    /* zero all of the accounts */
    for (i = 0; i < MAXACCTS; i++) {
        accounts[i] = 0;
    };

    /* create an audit object */
    INVOKE ObjMngr.Create(Audit, 1, Audt);
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(AUDIT_CREATE);

    /* initialize the audit object */
    INVOKE Audt.Init();
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(AUDIT_INIT);

    /* create a semaphore */
    INVOKE SemMngr.Create(Sem, 1);
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(SEM_CREATE);

    /* return a success indication */
    RETURN SUCCESS;
};
```

```
OPERATION  Deposit(IN int:acct, int:amt)
/*
 * Deposit the given amount in the specified account.
 */
{
    boolean    error = FALSE;

    /* begin a transaction */
    INVOKE TransMngr.Begin(BASIC);

    /* get exclusive access to the account database */
    INVOKE Sem.P();

    /* add to the given account */
    accounts[acct] += amt;

    /* release access to the account database */
    INVOKE Sem.V();

    /* log this in the audit trail */
    INVOKE Audt.Add(acct, DEPOSIT, amt);
    if (RESULT != SUCCESSFUL) {
        error = TRUE;
        INVOKE TransMngr.Abort();
    };

    /* commit the transaction */
    INVOKE TransMngr.End();

    /* determine outcome of transaction */
    if (error == TRUE) {
        /* return a failure indication */
        RETURN FAILURE(AUDIT);
    }
    else {
        /* return a success indication */
        RETURN SUCCESS;
    };
};


OPERATION  Withdraw(IN int:acct, int:amt)
/*
* Withdraw the given amount from the specified account.
*/
{
    boolean    error = FALSE;

    /* begin a transaction */
    INVOKE TransMngr.Begin(BASIC);

    /* get exclusive access to the account database */
    INVOKE Sem.P();

    /* remove from the given account */
    accounts[acct] -= amt;

    /* release access to the account database */
    INVOKE Sem.V();

    /* log this in the audit trail */
    INVOKE Audt.Add(acct, DEPOSIT, amt);
    if (RESULT != SUCCESSFUL) {
        error = TRUE;
```

```
        INVOKE TransMngr.Abort();
    };

    /* commit the transaction */
    INVOKE TransMngr.End();

    /* determine outcome of transaction */
    if (error == TRUE) {
        /* return a failure indication */
        RETURN FAILURE(AUDIT);
    }
    else {
        /* return a success indication */
        RETURN SUCCESS;
    };
};


OPERATION  Query(IN int:acct; OUT int:amt)
/*
* Query the amount in the specified account.
*/
{
    boolean    error = FALSE;

    /* begin a transaction */
    INVOKE TransMngr.Begin(BASIC);

    /* get exclusive access to the account database */
    INVOKE Sem.P();

    /* return the amount in the given account */
    amt = accounts[acct];

    /* release access to the account database */
    INVOKE Sem.V();

    /* log this in the audit trail */
    INVOKE Audt.Add(acct, DEPOSIT, amt);
    if (RESULT != SUCCESSFUL) {
        error = TRUE;
        INVOKE TransMngr.Abort();
    };

    /* commit the transaction */
    INVOKE TransMngr.End();

    /* determine outcome of transaction */
    if (error == TRUE) {
        /* return a failure indication */
        RETURN FAILURE(AUDIT);
    }
    else {
        /* return a success indication */
        RETURN SUCCESS;
    };
};
```

```
/*****************************************************************************\
*                                                                             *
*                                    AUDIT.C                                  *
*                                                                             *
\*****************************************************************************/

/*
 * Object Declaration
 */

OBJECT(Audit: PERMANENT, ATOMIC-UPDATE)


/*
 * Include Files
 */

#include        "console.h"


/*
 * Declarations
 */

#define MAXLOGS         100

typedef struct log {
        int     acct;
        cmd_t   cmd;
} log_t;

int     current;

log_t   logs[MAXACCTS];

WELLKNOWN CAPA LockManager      LckMngr;
CAPA           LogLock;

#define SUCCESSFUL    0
```

```
/*
 * Operations
 */

OPERATION  Init()
/*
 * Initialize the log object.
 */
{
    /* initialize the current log pointer */
    current = 0;

    /* allocate a lock for the log data */
    INVOKE LckMngr.Create(logs, MAXACCTS, LogLock);
    if (RESULT != SUCCESSFUL)
        RETURN FAILURE(LOCK_CREATE);

    /* return a success indication */
    RETURN SUCCESS;
};



OPERATION  Add(IN int:acct, int:amt, cmd_t:cmd)
/*
 * Add an operation to the audit trail log.
 */
{
    /* lock the log */
    INVOKE LogLock.Lock(ExclusiveWrite);

    /* add to the log */
    logs[current].acct = acct;
    logs[current].cmd = cmd;
    logs[current].amt = amt;

    /* bump the pointer, and roll it if necessary */
    current++ %= MAXLOGS;

    /* unlock the log */
    INVOKE LogLock.UnLock();

    /* return a success indication */
    RETURN SUCCESS;
};
```

References

[Ada 83] United States Department of Defense.
Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983.
Springer-Verlag, New York, 1983.

[Allchin 83] Allchin, J. E.
Support for Objects and Actions in Clouds.
School of Information and Computer Science, Project Report, Georgia Institute of Technology, May, 1983.

[Allen 85] Allen, L. W.
Design of a Kernel for Argus.
Technical Report 43, Massachusetts Institute of Technology, June, 1985.
Programming Methodology Group Memo.

[Almes 85] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D.
The Eden System: A Technical Review.
IEEE Transactions on Software Engineering SE-11(1):43-58, January, 1985.

[Anderson 81] Anderson, T. and Lee, P. A.
Fault Tolerance: Principles and Practice.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Applewhite 81] Applewhite, H. L., Jensen, E. D., Northcutt, J. D, Sha, L. and Tokoro M.
Distributed and Hierarchical VLSI Computer Architecture for Real-Time Applications.
1981
Final Report to RCA, Department of Electrical Engineering and Department of Computer Science, Carnegie-Mellon University.

[Avizienis 78] Avizienis, A.
Fault-Tolerance: The Survival Attribute of Digital Systems.
Proceedings of the IEEE 66(10):1109-1125, October, 1978.

[Baskett 77] Baskett, F., Howard, J. H. and Montague, J. T.
Task Communication in DEMOS.
Operating Systems Review 11(5):23-32, November, 1977.

[Bayer 79] Bayer, R., Graham, R. M. and Seegmueller, G. (editors).
Lecture Notes in Computer Science. Volume 60: *Operating Systems: An Advanced Course.*
Springer-Verlag, Berlin, 1979.

[Bechtolsheim 82] Bechtolsheim, A., Baskett, F. and Vaughan, P.
The SUN Workstation Architecture.
Technical Report 229, Computer System Laboratory, Stanford University, March, 1982.

[Bernstein 81] Bernstein, P. A. and Goodman, N.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[Berstis 80] Berstis, V.
Security and Protection of Data in the IBM System/38.
In *Proceedings of the Seventh Symposium on Computer Architecture*, pages 245-252. IEEE, May, 1980.

[Bochmann 76] Bochmann, G. V.
Finite-State Description of Communication Protocols.
Technical Report 236, Departement d'Informatique, Universite de Montreal, July, 1976.

[Boebert 78] Boebert, W. E.
Concepts and Facilities of the HXDP Executive.
Technical Report 78SRC21, Honeywell Systems & Research Center, March, 1978.

[Boehm 81] Boehm, B. W.
Advances in Computer Science and Technology: Software Engineering Economics.
Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[Brinch Hansen 70]
Brinch Hansen, P.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(4):238-250, April, 1970.

[Brinch Hansen 71]
Brinch Hansen, P.
RC4000 Software Multiprogramming System.
A/C Regnecentralen, Copenhagen, 1971.

[Burke 83] Burke, E.
The CMOS Operating System.
1983
Bolt, Beranek and Newman, Internal Report.

[Cheriton 84a] Cheriton, D. R.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software 1(2):19-43, January, 1984.

[Cheriton 84b] Cheriton, D. R. and Zwaenepoel, W.
One-to-Many Interprocess Communication in the V-System.
In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 64-80. ACM, June, 1984.

[Clark 83] Clark, R. K. and Shipman, S. E.
The Archons Testbed — A Requirements Study.
Archons Project Internal Document, Department of Computer Science, Carnegie-Mellon University, May, 1983.

[Clark 87] Clark, R. K.
Operating System Kernel Support for Compound Atomic Transactions.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1987.
In progess.

[Colwell 85] Colwell, R. P.
The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, August, 1985.

[Cooper 84] Cooper, E. C.
Replicated Procedure Call.
In *Proceedings of Third Annual Symposium on Principles of Distributed Computing*, pages 220-232. ACM, August, 1984.

[Cox 83] Cox, G. W., Corwin, W. M., Lai, K. K. and Pollack, F. J.
Interprocess Communication and Processor Dispatching on the Intel 432.
ACM Transactions on Computer Systems 1(1):45-66, February, 1983.

[Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.

[Danthine 80] Danthine, A. A. S.
Protocol Representation with Finite-State Models.
IEEE Transactions on Communications COM-28(4):632-643, April, 1980.

[Dennis 66] Dennis, J. B. and Van Horn, E. C.
Programming Semantics for Multiprogrammed Computation.
Communications of the ACM 9(3):143-155, March, 1966.

[DRC 86] Dynamics Research Corporation.
Distributed Systems Technology Assesment for SDI.
Technical Report E-12256U, Electronic Systems Division, United States Air Force Systems Command, September, 1986.

[Eastport 85] Eastport Study Group.
A Report to the Director Strategic Defense Initiative Organization.
Summer Study 1985, United States Department of Defense, December, 1985.

[Eswaran 76] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[Fabry 74] Fabry, R. S.
Capability-Based Addressing.
Communications of the ACM 17(7):403-412, July, 1974.

[Farber 72] Farber, D. J. and Larson, K. C.
The System Architecture of the Distributed Computer System — The Communications System.
In *Proceedings, Symposium on Computer-Communications Networks and Teletraffic*, pages 21-27. Polytechnic Institute of Brooklyn, April, 1972.

[Fitzgerald 85] Fitzgerald, R. and Rashid R.
The Integration of Virtual Memory Management and Interprocess Communication in Accent.
In *Proceedings, Tenth Symposium on Operating Systems Principles*, pages 13-14. ACM, November, 1985.

[Fletcher 78] Fletcher, J. G. and Watson, R. W.
Mechanisms for a Reliable Timer-Based Protocol.
In *Proceedings, Computer Network Protocols*, pages C5-1 — C5-17. Universite' De Liege, February, 1978.

[Fletcher 79] Fletcher, J. G.
Serial Link Protocol Design: A Critique of the X.25 Standard — Level 2.
Technical Report UCRL 83604, Lawrence Livermore Laboratory, August, 1979.

[Forsdick 78] Forsdick, H. C., Schantz, R. E., Thomas, R. H.
Operating Systems for Computer Networks.
Computer 11(1):48-57, January, 1978.

[Franta 81] Franta, W. R., Jensen, E. D., Kain, R. Y. and Marshall, G. D.
Real-Time Distributed Computer Systems.
Advances in Computers 20:39-82, 1981.

[Garcia-Molina 83] Garcia-Molina, H.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transaction on Database Systems 8(2), June, 1983.

[Geller 77] Geller, D. P.
The National Software Works: Access to Distributed Files and Tools.
In *Proceedings, ACM National Conference*, pages 39-43. October, 1977.

[Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 150-162. ACM, December, 1979.

[Glass 80] Glass, R. L.
Real-Time: The 'Lost World' of Software Debugging and Testing.
Communications of the ACM 23(5):264-271, May, 1980.

[Goldberg 83] Goldberg, A. and Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, Massachusetts, 1983.

[Goodman 83] Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S. and Ries, D.
A Recovery Algorithm for a Distributed Database System.
In *Proceedings, Second Symposium on Principles of Database Systems.* ACM, March, 1983.

[Habermann 76] Habermann, A. N., Flon, L. and Cooprider, L.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM 19(5):266-272, May, 1976.

[Herlihy 85] Herlihy, M. P.
Using Type Information to Enhance the Availability of Partitioned Data.
Technical Report CMU-CS-85-119, Department of Computer Science, Carnegie-Mellon University, April, 1985.

[Herlihy 86] Herlihy, M. P.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[Hoare 74] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[Jensen 76] Jensen, E. D. and Anderson, G. A.
Feasibility Demonstration of Distributed Processing for Small Ships Command and Control Systems.
Final Report N00123-74-C-0891, Honeywell Systems & Research Center, August, 1976.

[Jensen 78a] Jensen, E. D.
The Honeywell Experimental Distributed Processor — An Overview.
Computer 11(1):137-147, January, 1978.

[Jensen 78b] Jensen, E. D., Marshall, G. D., White, J. A. and Helmbrecht, W. F.
The Impact of Wideband Multiplex Concepts on Microprocessor-Based Avionic System Architectures.
Technical Report AFAL-TR-78-4, Honeywell Systems & Research Center, February, 1978.

[Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System.
In *Distributed Processing Technical Committee Newsletter.* IEEE, June, 1984.
Special Issue on Distributed Operating Systems.

[Jones 79] Jones, A. K., Chansler Jr., R. J., Durham, I., Schwans, K. and Vegdahl, S. R.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating Systems Principles,* pages 117-127. ACM, December, 1979.

[Kahn 81] Kahn, K. C., Corwin, W. M., Dennis, T. D., Hooge, H. D., Hubka, D. E. and Hutchins, L. A.
iMAX: A Multiprocessing Operating System for an Object-Based Computer.
In *Proceedings, Eighth Symposium on Operating System Principles*, pages 127-136. ACM, December, 1981.

[Katsuki 78] Katsuki, D., Elasm, E. S., Mann, W. F., Roberts, E. S., Robinson, J. G., Skowronski, F. S. and Wolf, E. W.
Pluribus — An Operational Fault-Tolerant Multiprocessor.
Proceedings of the IEEE 66(10):1146-1159, October, 1978.

[Kung 81] Kung, H. T. and Robinson, J. T.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

[Lampson 69] Lampson, B. W.
Dynamic Protection Structures.
In *Proceedings of the Fall Joint Computer Conference*, pages 27-38. IFIPS, 1969.

[Lampson 76] Lampson, B. W. and Sturgis, H. E.
Reflections on an Operating System Design.
Communications of the ACM 19(5):251-265, May, 1976.

[Lampson 81] Lampson, B. W., Paul, M. and Siegert, H. J. (editors).
Lecture Notes in Computer Science. Volume 105: *Distributed Systems — Architecture and Implementation.*
Springer-Verlag, Berlin, 1981.

[Lechoczky 86] Lechoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review 14(1):44-53, May, 1986.

[Lehman 85] Lehman, M. M. and Belady, L. A.
Program Evolution: Processes of Software Change.
Academic Press, London, 1985.

[Leinbaugh 80] Leinbaugh, D. W.
Guaranteed Response Times in a Hard-Real-Time Environment.
IEEE Transactions on Software Engineering SE-6(1):85-91, January, 1980.

[Levin 75] Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W.
Policy/Mechanism Separation in Hydra.
In *Proceedings, Fifth Symposium on Operating Systems Principles*, pages 132-140. ACM, November, 1975.

[Levin 77] Levin, R.
Program Structures for Exceptional Condition Handling.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, June, 1977.

[Levy 84] Levy, H. M.
Capability-Based Computer Systems.
Digital Press, Bedford, Massachusetts, 1984.

[Liskov 84] Liskov, B. H.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February, 1984.

[Liskov 85] Liskov, B. H., Herlihy, M. P., Gilbert, L.
Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing.
Technical Report CMU-CS-85-168, Department of Computer Science, Carnegie-Mellon University, October, 1985.

[Lister 77] Lister, A.
The Problem of Nested Monitor Calls.
ACM Operating System Review 11(2):5-7, July, 1977.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, May, 1986.

[McKendry 84a] McKendry, M. S.
The Clouds Project: Reliable Operating Systems for Multicomputers.
Project Report, Georgia Institute of Technology, 1984.

[McKendry 84b] McKendry, M. S.
Ordering Actions for Visibility.
Project Report GIT-ICS-84/05, Georgia Institute of Technology, February, 1984.

[McKendry 85] McKendry, M. S. and Herlihy, M. P.
Time-Driven Orphan Elimination.
Department of Computer Science CMU-CS-85-138, Department of Computer Science, Carnegie-Mellon University, 1985.

[McQuillan 80] McQuillan, J. M., Richer, I. and Rosen, E. C.
The New Routing Algorithm for the ARPANET.
IEEE Transactions on Communications COM-28(5):711-719, May, 1980.

[Metcalf 72] Metcalf, R. M.
Strategies for Interprocess Communication in a Distributed Computing System.
In *Proceedings, Symposium on Computer-Communications Network and Teletraffic,* pages 519-526. Polytechnic Institute of Brooklyn, April, 1972.

[Mockapetris 77] Mockapetris, P. V., Lyle, M. and Farber, D. J.
On the Design of a Local Network Interface.
In *Proceedings, Information Processing 77.* IFIP, 1977.

[Morris 73] Morris, J. H.
Protection in Programming Languages.
Communications of the ACM 16(1):15-21, January, 1973.

[Moss 85] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
The MIT Press, Cambridge, Massachusetts, 1985.

[Nelson 81] Nelson, B. J.
Remote Procedure Call.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, May, 1981.

[Ousterhout 79] Ousterhout, J. K., Scelza, D. A. and Sindhu, P. S.
Medusa: An Experiment in Distributed Operating System Structure (Summary).
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 115-116. ACM, December, 1979.

[Ousterhout 80] Ousterhout, J. K., Scelza, D. A. and Sindhu, P. S.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.

[Parnas 77] Parnas, D. L.
Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.
Technical Report 8047, Naval Research Laboratory, December, 1977.

[Pease 80] Pease, M., Shostak, R. and Lamport, L.
Reaching Agreement in the Presence of Faults.
Journal of the ACM 27(2):228-234, 1980.

[Plummer 82] Plummer, D. C.
An Ethernet Address Resolution Protocol.
RFC 826.
November, 1982

[Popek 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proceedings of the 8th. Symposium on Operating Systems Principles*, pages 160-168. ACM, December, 1981.

[Quirk 85] Quirk, A. B.
Verification and Validation of Real-Time Software.
Springer-Verlag, Berlin, 1985.

[Randell 78] Randell, B., Lee, P. A. and Treleaven, P. C.
Reliability Issues in Computing System Design.
Computing Surveys 10(2):123-165, June, 1978.

[Rashid 81] Rashid, R. F. and Robertson, G. G.
Accent: A Communication Oriented Network Operating System Kernel.
Technical Report 123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Ready 86] Ready, J. F.
VRTX: A Real-Time Operating System for Embedded Microprocessor Applications.
IEEE Micro 6(4):8-17, August, 1986.

[Savitzky 85] Savitzky, S. R.
Real-Time Microprocessor Systems.
Van Nostrand Reinhold, New York, 1985.

[Schantz 85] Schantz, R., Schroder, M., Barrow, M., Bono G., Dean M., Gurwitz, R., Lebowitz, K. and Sands, R.
CRONUS, A Distributed Operating System: Interim Technical Report No. 5.
Technical Report 5991, Bolt Beranek and Newman, June, 1985.

[Sha 85a] Sha, L.
Modular Concurrency Control and Failure Recovery — Consistency, Correctness and Optimality.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[Sha 85b] Sha, L., Lehoczky, J. P. and Jensen, E. D.
Modular Concurrency Control and Failure Recovery.
Technical Report, Carnegie-Mellon University, November, 1985.
Archons Project Report.

[Shipman 87] Shipman, S. E.
Mechanisms to Support Object Replication for a Distributed Kernel.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1987.
In progess.

[Smith 79] Smith, R. G.
The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.
In *Proceedings, First International Conference on Distributed Computing*, pages 185-192. IEEE, October, 1979.

[Spector 84] Spector, A. H.
Support for Distributed Transactions in the TABS Prototype.
Technical Report CMU-CS-84-132, Department of Computer Science, Carnegie-Mellon University, 1984.

[Sturgis 80] Sturgis, H., Mitchell, J., and Israel, J.
Issues in the Design and Use of a Distributed File System.
ACM Operating Systems Review 14(3):55-69, July, 1980.

[Sun 82] Rattner, M., Bechtolsheim, A., Gilmore, J., Joy, B., Lyon, T., McGilton, H., and Shannon, B.
Programmer's Reference Manual for the Sun Workstation.
Technical Report 800-0345, Sun Microsystems, Inc., October, 1982.

[Sun 84] Sun Microsystems.
Engineering Manual for the Sun-2/120 CPU Board.
Technical Report 800-1185-01, Sun Microsystems, Inc., September, 1984.

[Thomas 78] Thomas, R. H., Schantz, R. E. and Forsdick, H. C.
Network Operating Systems.
Technical Report 3796, Bolt, Beranek and Newman, 1978.

[Thompson 80] Thompson, J. R., Ruspini, E. H. and Montgomery, C. A.
TAC C^3 Distributed Operating System Study.
Technical Report RADC-TR-79-360, Operating Systems, Inc. for Rome Air Development Center, January, 1980.

[Traiger 82] Traiger, I.
Virtual Memory Management for Database Systems.
ACM Operating Systems Review 16(4), October, 1982.

[Wilkes 79] Wilkes, M. V. and Needham, R. M.
The Cambridge CAP Computer and its Operating System.
North Holland, New York, 1979.

[Wirth 77] Wirth, N.
Towards a Discipline of Real-Time Programming.
Communications of the ACM 20(8):577-585, August, 1977.

[Wittie 79] Wittie, L. D.
A Distributed Operating System For a Reconfigurable Network Computer.
In *Proceedings, First International Conference on Distributed Computing Systems,* pages 669-677. IEEE, October, 1979.

[Wulf 81] Wulf, W. A., Levin, R. and Harbison, S. P.
Hydra/C.mmp: An Experimental Computer System.
McGraw/Hill, New York, 1981.

PERSPECTIVES IN COMPUTING

Vol. 1 John R. Bourne, *Laboratory Minicomputing*

Vol. 2 Carl Tropper, *Local Computer Network Technologies*

Vol. 3 Kendall Preston, Jr., and Leonard Uhr, editors, *Multicomputers and Image Processing: Algorithms and Programs*

Vol. 4 Stephen S. Lavenberg, editor, *Computer Performance Modeling Handbook*

Vol. 5 R. Michael Hord, *Digital Image Processing of Remotely Sensed Data*

Vol. 6 Sakti P. Ghosh, Y. Kambayashi, and W. Lipski, editors, *Data Base File Organization: Theory and Applications of the Consecutive Retrieval Property*

Vol. 7 Ulrich W. Kulisch and Willard L. Miranker, editors, *A New Approach to Scientific Computation*

Vol. 8 Jacob Beck, Barabara Hope, and Azriel Rosenfeld, editors, *Human and Machine Vision*

Vol. 9 Edgar W. Kaucher and Willard L. Miranker, *Self-Validating Numerics for Function Space Problems: Computation with Guarantees for Differential and Integral Equations*

Vol. 10 Mohamed S. Abdel-Hameed, Erhan Çınlar, and Joseph Quinn,

Volumes 1 – 12 were published as **Notes and Reports in Computer Science and Applied Mathematics.**

editors, *Reliability Theory and Models: Stochastic Failure Models, Optimal Maintenance Policies, Life Testing, and Structures*

Vol. 11 Mark G. Karpovsky, editor, *Spectral Techniques and Fault Detection*

Vol. 12 Selim G. Akl, *Parallel Sorting Algorithms*

Vol. 13 Azriel Rosenfeld, editor, *Human and Machine Vision II*

Vol. 14 Y. C. Tay, *Locking Performance in Centralized Databases*

Vol. 15 David S. Johnson, Takao Nishizeki, Akihiro Nozaki, Herbert S. Wilf, editors, *Discrete Algorithms and Complexity: Proceedings of the Japan-US Joint Seminar, June 4–6, 1986, Kyoto, Japan*

Vol. 16 J. Duane Northcutt, *Mechanics for Reliable Distributed Real-Time Systems: The Alpha Kernel*