The Alpha Operating System: Kernel Internals

J. Duane Northcutt
Raymond K. Clark

TR #88051

Archons Project Technical Report

May 29, 1988

The Alpha Operating System: Kernel Internals

J. Duane Northcutt
Raymond K. Clark

May 29, 1988

Archons Project Technical Report #88051

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

List of Figures

Abstract

This report describes the design and implementation of the kernel layer of Release 1 of the Alpha operating system.

Alpha is an adaptable decentralized operating system for real-time applications, being developed as a part of the Archons project's on-going research into real-time distributed systems. Alpha is unique in two particularly significant ways—first, it is *decentralized,* providing reliable resource management transparently across physically dispersed nodes, so that a distributed application can be implemented and executed as though it were centralized; and second, it provides comprehensive, high technology support for real-time applications, particularly supervisory control systems (e.g., industrial automation applications), which are characterized by predominately aperiodic activities executing under critical time constraints (such as deadlines). Alpha is extremely adaptable so as to be easily optimized for a wide range of problem-specific functionality, performance, and cost.

Alpha has been designed to support real-time, reliable, distributed applications. Since Alpha is, in fact, an example of just such an application, it has been designed in a layered manner. The lowest layer, the *kernel*, supports the fundamental abstractions of the Alpha programming model [Northcutt 88b]. The system layer of the operating system, which is based on the kernel, can then utilize the power of this programming model, as well as the user's application program.

In this report, details are given concerning the design and implementation of the major facilities provided by the Alpha kernel. This includes support for the object, thread and invocation abstractions, as well as the facilities for inter-node communication management, virtual memory management, application processor management, and secondary storage management.

1 Introduction

This report describes the design and implementation of the kernel layer of the Alpha operating system. The kernel of Alpha provides support for the Alpha programming model (as defined in [Northcutt 88b]) by implementing the kernel interface (defined in [Northcutt 88d]). The overall system context within which the kernel layer of Alpha exists and the various functions provided at each layer are illustrated in Figure 1.

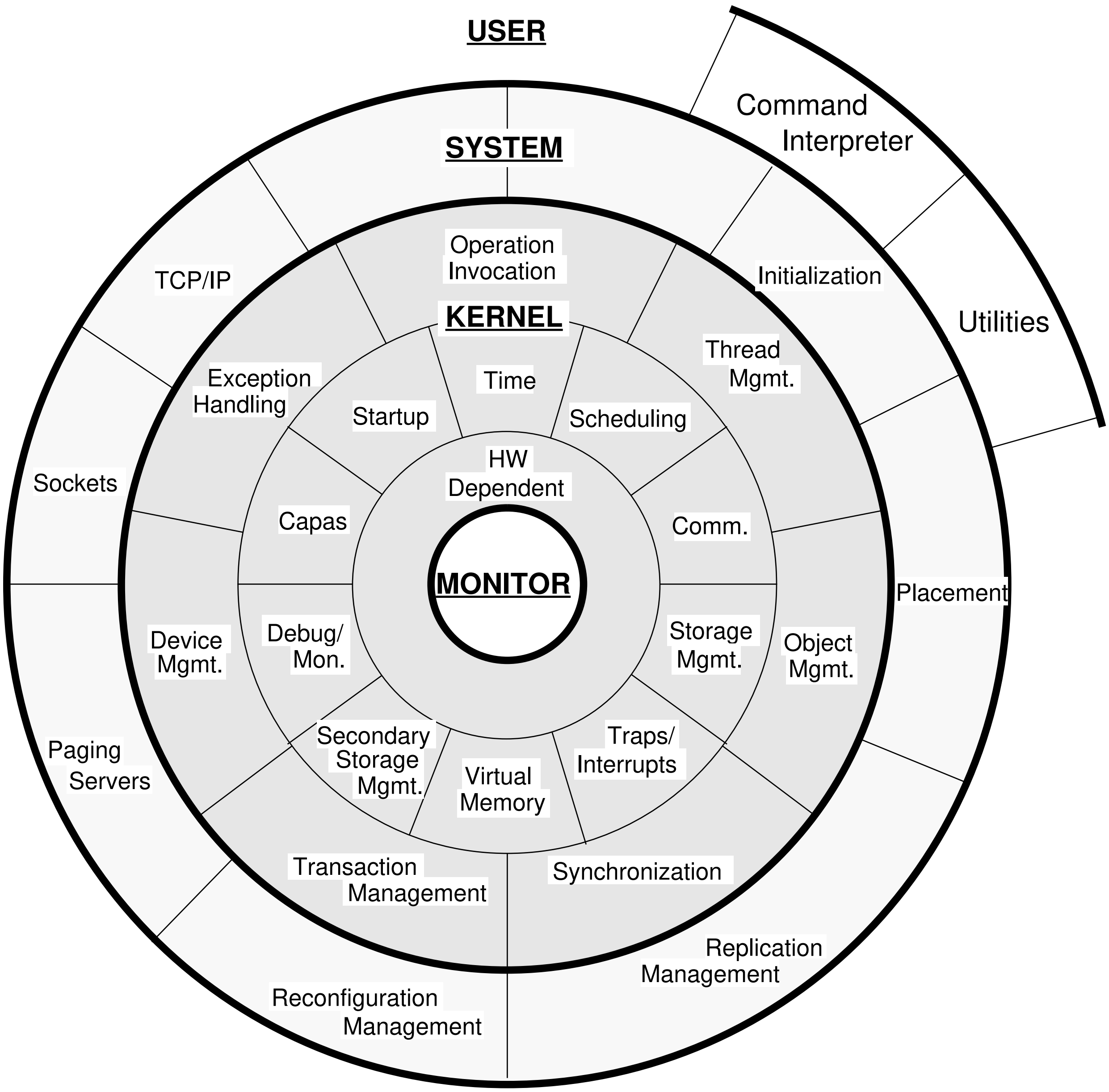


Figure 1: System Functional Design Structure

Alpha has been designed to support real-time, reliable, distributed applications. Since Alpha is, in fact, an example of just such an application, it has been designed in a layered manner. The lowest layer, the *kernel*, supports the fundamental abstractions of the Alpha programming model [Northcutt 88b]. The system layer of the operating system, which is based on the kernel, can then utilize the power of this programming model, as well as the user's application program.

The Alpha kernel layer is composed primarily of C language routines, with a few assembly language routines. The kernel routines are divided into two groups: routines that provide kernel services to clients by means of an object interface, and basic support routines, that provide services needed for the kernel-level abstractions, but that are not visible to the kernel's clients. Kernel routines are grouped according to their logical function, corresponding to kernel *facilities*. 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: object management, thread management, operation invocation, 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 locally—i.e., on the node at which they exist. Each of these mechanisms provides its 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 communication, secondary storage, and peripheral devices.

This report describes details of the design and implementation of the major facilities provided by the Alpha kernel. This includes support for the object, thread and invocation abstractions, as well as the facilities for 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 performed by a replica of the kernel on a single node. Furthermore, the structure of this report parallels that of the kernel itself—i.e., each of the kernel's major structural components are discussed in a separate chapter. Figure 2 provides an illustration of the internal structure of a replica of the kernel.

The following description of the kernel is given in the context of the hardware described in [Northcutt 88c]. Machine dependencies exist in most operating system kernels, and the Alpha kernel is no exception. However, although machine dependencies can be found in 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 processors supporting either a linear, or a segmented, virtual address space and demand-paged virtual memory.

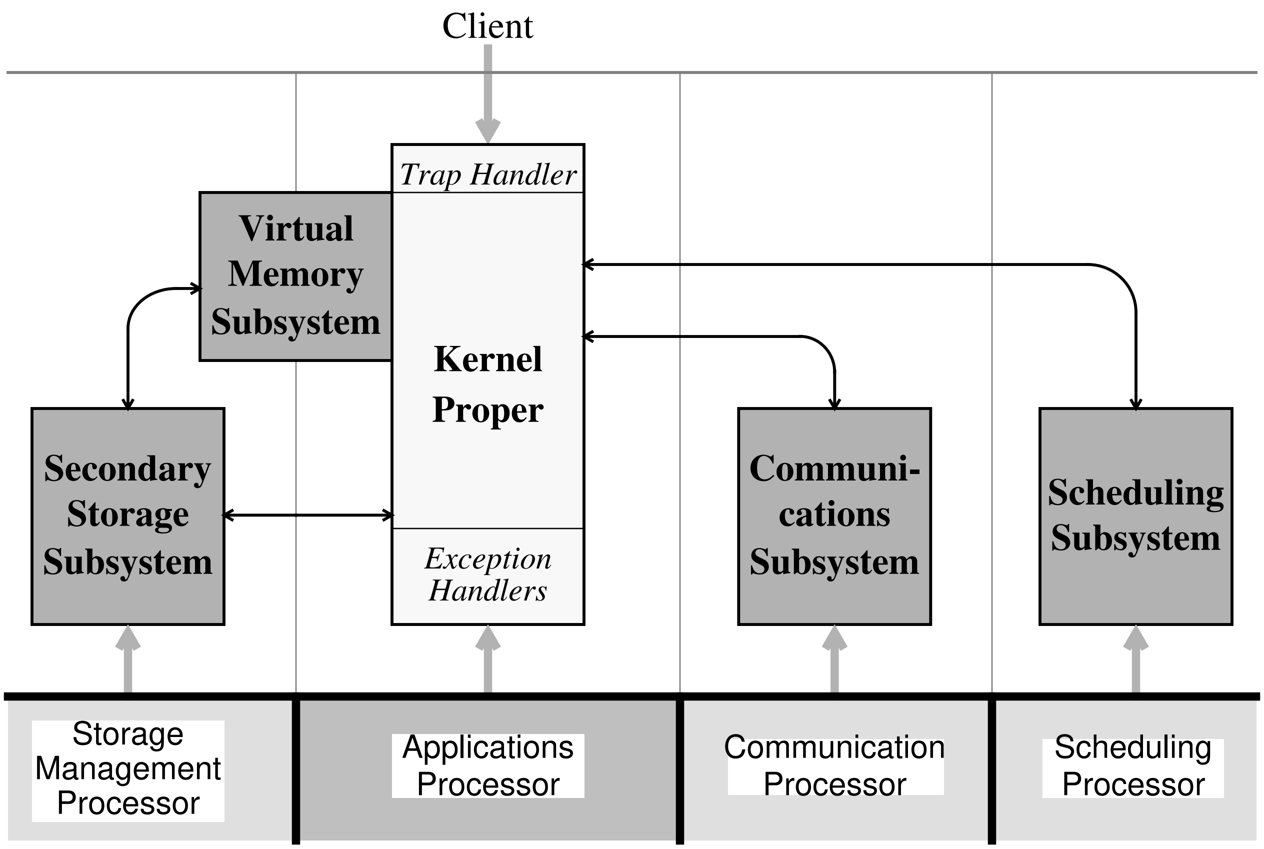


Figure 2: Intra-Node Kernel Structure

2 Kernel Structure

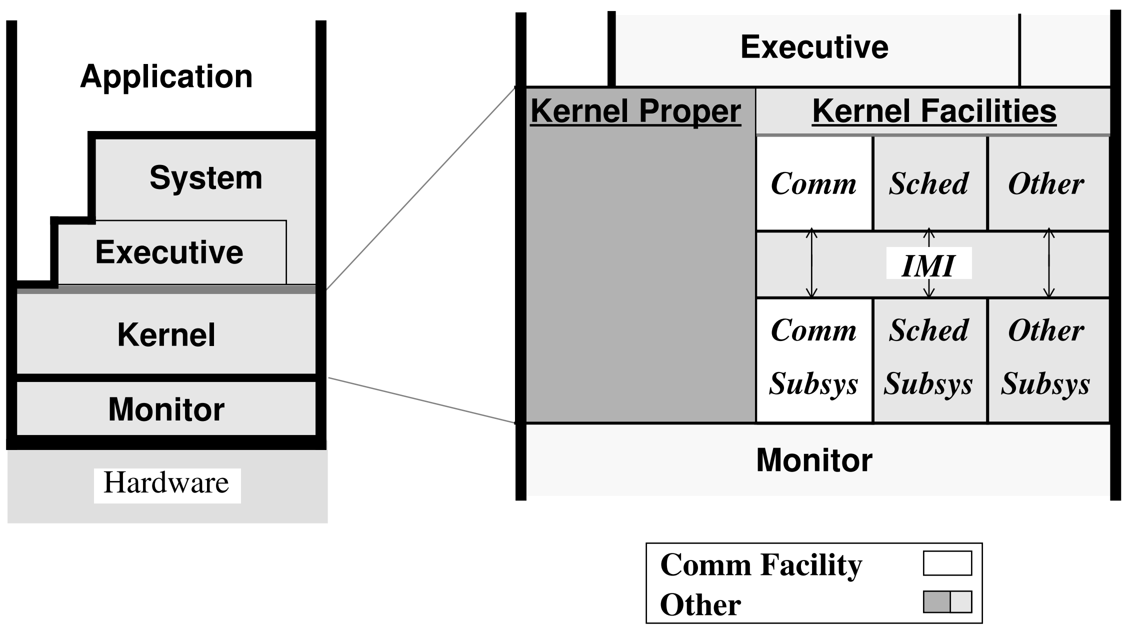
The logical structure of the Alpha Operating System ([Northcutt 87]) is illustrated in Figure 3. Applications run on top of the Alpha Operating System, which internally has the following components:

1. **Kernel** — which supports the basic abstractions of objects, threads, and invocations;
2. **Executive and System** — which offer higher-level operating system functions and embody global resource management policies; both the Executive and System are comprised of objects and threads, and the distinction between them is unimportant for this discussion.

The Alpha Kernel executes on top of a small layer of software, called the Monitor, that provides the lowest level hardware access, allowing the execution of primitive operations and minimal debugging.

The internal structure of the Alpha Kernel, shown in Figure 3b, features two major parts:

- the *Kernel Proper* and
- a set of *Kernel Facilities*.



(a) General Structure (b) Kernel Internal Structure

Figure 3: Logical Alpha System Structure

The Kernel Facilities provide important kernel services and are particularly interesting because their services may typically execute concurrently. Consequently, each of the facilities is composed of two parts:

- one of which executes on the Applications Processor along with the Kernel Proper, and
- another which executes on a separate co-processor.

- The part of a kernel facility that executes on a separate co-processor is called a *Kernel Subsystem*. Every kernel facility has a corresponding Kernel Subsystem. For example, in Figure 3b, the Communication Facility, which has been shown in white, while the rest of the Kernel Proper and the other kernel facilities have been shaded, consists of two parts, one of which is labeled the Communication Subsystem..

The Alpha Kernel structure provides actual concurrency for operating system tasks— the Kernel Proper and all of the Kernel Subsystems can operate simultaneously. For instance, the Kernel Proper can perform a local invocation while the Communication Subsystem processes a remote invocation and monitors the current health of all of the threads rooted on the local node. At the same time, the Scheduling Subsystem can revise the execution schedule, and the Storage Management Subsystem can locate and acquire the initial load image for the creation of a new object instance.

The functional partitioning of the kernel into the Kernel Proper and a number of Kernel Facilities that feature independent Kernel Subsystems also facilitates the use of dedicated, special purpose hardware that may be selected and optimized to perform only those functions that are required of a given subsystem, rather than a set of general purpose tasks.

Each Kernel Subsystem communicates with the portion of its facility that resides on the Application Processor with the Kernel Proper by means of a dedicated, two-way, communication channel, known as an *Interprocessor Message Interface* (IMI) channel. (Some subsystems, such as the Communication Subsystem, also share a limited amount of memory with the Kernel Proper.) The IMI channel, which is described in detail below, permits one processor to send a message, containing a command or data, to another processor. In addition, the sender may optionally interrupt the receiver to notify it of the presence of a new message—thus providing an efficient delivery mechanism.

2.1 Alpha Hardware Architecture

Alpha is intended to execute on a set of nodes interconnected by one or more global communication buses. (Currently, the Alpha Experimental Testbed uses an Ethernet as its single global bus.) Since there are a number of operating system activities that can benefit from genuine hardware concurrency—for instance, communication processing and scheduling analysis, in addition to application execution—each testbed node contains multiple processors. As shown in Figure 4, each node has an Application Processor (AP), a Scheduling Co-Processor (SP), a Communication Co-Processor (CP), an Ethernet Controller Board, and a pool of shared memory, all connected by a Multibus backplane. In addition, other co-processors may optionally be added—for instance, to manage primary and secondary storage, a Storage Management Co-Processor (SMP) and one or more Disk Controller Boards may be added. (For this purpose, Alpha currently supports Fujitsu MK2284 168 Megabyte disk drives controlled by a Xylogics 450 disk controller.)

In the current version of the Alpha Experimental Testbed, the AP is a Sun Microsystems Inc. (SMI) version 2.0 CPU board, while all of the co-processors are SMI version 1.5 CPUs. The Ethernet Controller Board, also from SMI, is used to access the Ethernet. It is based on the Intel 82586 Local Area Network (LAN) Co-Processor and has 256 Kilobytes of on-board memory for use by the 82586. In addition, there are 256 Kilobytes of Multibus Shared Memory.

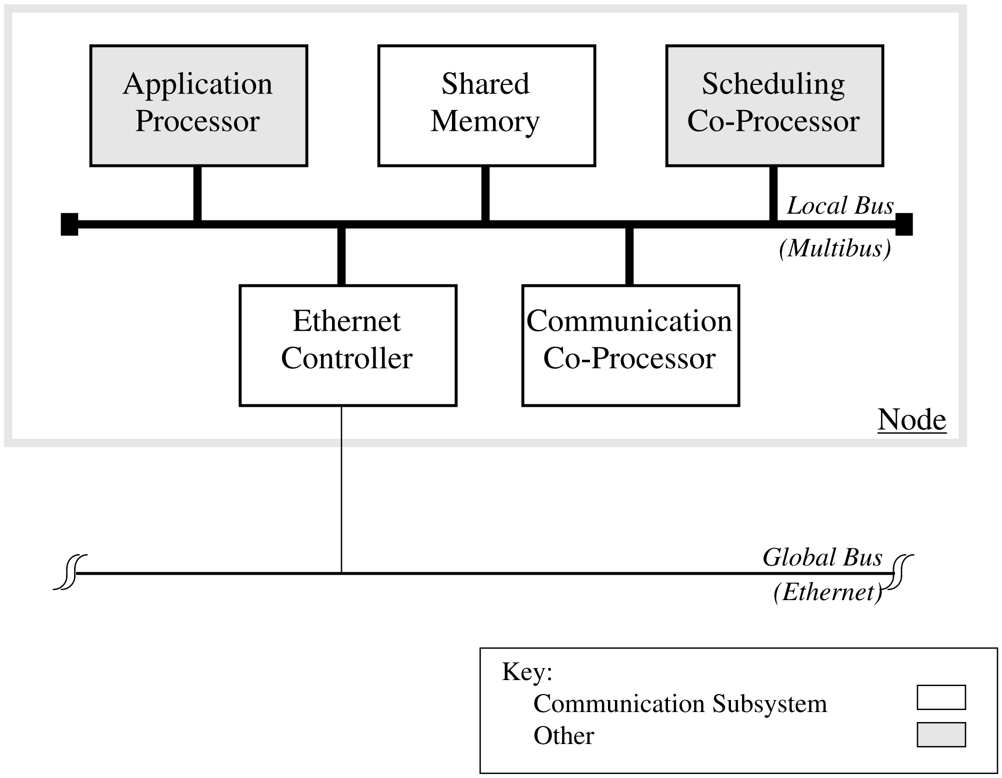


Figure 4: Alpha Experimental Testbed Node Architecture

While the testbed architecture provides a rich base for experimentation, it does impose a set of requirements and constraints on Alpha. Specifically, an Ethernet must interconnect the Alpha nodes, and this Ethernet must be controlled by the SMI Ethernet Controller Board. Furthermore, only off-the-shelf hardware can be considered if any additional hardware is required to provide the desired communication architecture.

As shown in Figure 4, the Communication Subsystem hardware consists of the CP, the Ethernet Controller Board, and the Multibus Shared Memory. All of the other boards are associated with other subsystems or the AP.

2.2 Kernel Proper

Ideally, the Application Processor(s) spends all of its time executing application code— that is, code that is specific to the application at hand, not code that is devoted to maintaining operating system data structures, performing operation invocations, scheduling threads, managing virtual memory mappings, or controlling disks drives or other peripherals. In such an ideal situation, the operating system functions are provided by one or more operating system processors.

Of course, this ideal system cannot be completely realized. At the very least, a relatively small amount of code must reside on the Application Processor to allow it to request operating system services from the operating system processors and to allow it to react to messages received from those processors. It may also be more efficient to provide some operating system functions directly on the AP if the communication and coordination overhead incurred in using multiple processors was too great.

Alpha aspires to provide an ideal system in the terms just described. However, due to the hardware available when Alpha was begun and the immense technological leap to provide an ideal system, a short-term compromise was reached. Currently, while a number of important operating system functions are run on dedicated processors, many other system functions are provided on the Application Processor. Although this might suggest that the operating system has not been completely separated from the application, it can more appropriately be viewed as a reflection of the fact that the Alpha Experimental Testbed has no Application Processor. To remedy that situation, one or more additional processors, depending on the demands of the application, should be added to each testbed node to act as true, dedicated Application Processors.

For Alpha Release 1, the Kernel Proper executes on the each node's Application Processor, along with a portion of each kernel facility that is provided on that node.

Since a detailed description of the Kernel Proper is offered in the following chapter, a brief example will serve to illustrate the division of function between the software residing on the AP and that residing on a co-processor (corresponding to one of the Kernel Subsystems): the Communication Facility, which must meet the communication and distribution requirements outlined in Chapter 5, is composed of software residing in both the Application Processor and the Communication Co-Processor. The functions performed by the Communication Facility on the Application Processor include local invocation support (that is, support for invocations whose target is on the same node as the invoker), memory mapping manipulations and thread control operations to initiate and complete remote invocations, and exception handling for failed invocations on replicated objects. All of these functions require the use of information maintained by the Kernel Proper and unavailable to the Communication Subsystem directly. Consequently, they are performed by the Communication Facility software on the Application Processor, which does have access to this information.

2.3 Kernel Subsystems

Alpha currently has three subsystems:

- the Scheduling Subsystem,
- the Communication Subsystem, and
- the Storage Management Subsystem.

The standard Alpha node employs a Scheduling Co-Processor and a Communication Co-Processor to support their respective subsystems. Optionally, any of the nodes may also have a Storage Management Co-Processor. In principle, each node could contain a single processor that executes both Alpha and the application.

Future versions of Alpha may support more subsystems than the current version. They may also use more co-processors, possibly with specialized hardware support, to improve and expand on the services provided by the existing subsystems. For example, the services provided by the Storage Management Subsystem may be refined and enhanced so that the Storage Management Subsystem will be replaced by two subsystems—the Primary Storage Management Subsystem, which would manage primary memory, including services such as anticipating the primary memory needs of the local objects and threads to improve performance, and the Secondary Storage Management Subsystem, which would carry out disk management and reliability support functions.

The hardware used for the subsystems was described earlier in Section 2.1 and, in greater depth, in the document, *The Alpha Distributed Computer System Testbed* ([Northcutt 88c]).

For the present, in order to contrast the type of function carried out by a co-processor with that carried out by the Application Processor, consider once again the example of the Communication Subsystem: the Communication Subsystem accepts high-level commands from the Application Processor and performs them in parallel with the continued execution of the Application Processor. The Communication Subsystem handles per-packet interrupts and interrupts the Application Processor only to indicate the occurrence of high-level communication events (e.g., an incoming invocation arrival or an outgoing invocation completion). The specific functions provided by the Communication Subsystem include message disassembly and reassembly; participation in the intra- and inter-node thread maintenance protocols, the atomic transaction refresh protocols, the atomic transaction commit and abort protocols, and object replication protocols; and all of the low-level acknowledgment generation and handling.

In the following chapters, both the structure of, and the functions provided by, each of the subsystems will be described in detail.

2.4 IMI Facility

Processors in an Alpha node communicate by means of the Interprocessor Message Interface (IMI) facility. The Application Processor (AP) is connected to each of its node's co-processors by an IMI *channel*. As illustrated by Figure 5, each IMI channel is composed of a pair of unidirectional communication links, called *streams*. For each processor, one stream is designated the *out stream* and transfers IMI messages originated by that processor, while the other stream is designated the *in stream* and receives messages from the other processor on that channel.

Each stream queues IMI messages. A small queue proved to be necessary for Alpha Release 1 in order to avoid deadlock situations that occurred when one channel was used to send multiple messages rapidly between a pair of processors. Despite the presence of the queue, which relaxes the response time requirements that must be observed by a processor to avoid deadlock, it is still desired that the co-processors respond as rapidly as possible to IMI messages, which are typically commands, issued by the AP. Quick co-processor response allows the AP to dedicate itself to application code as much as possible, rather than forcing the AP to spend (relatively) large amounts of its time waiting on responses to IMI messages or waiting to send its next command to a co-processor, because of a full IMI stream queue. Consequently, all of the co-processors have been designed to respond rapidly to AP-originated IMI messages.

Interrupts are employed to provide good responsiveness to IMI message arrival without imposing a high on-going system overhead. The originator of an IMI message may use the *attention line* shown in Figure 5 to cause an interrupt to be sent to the message's receiver once the message has been placed in its out stream.

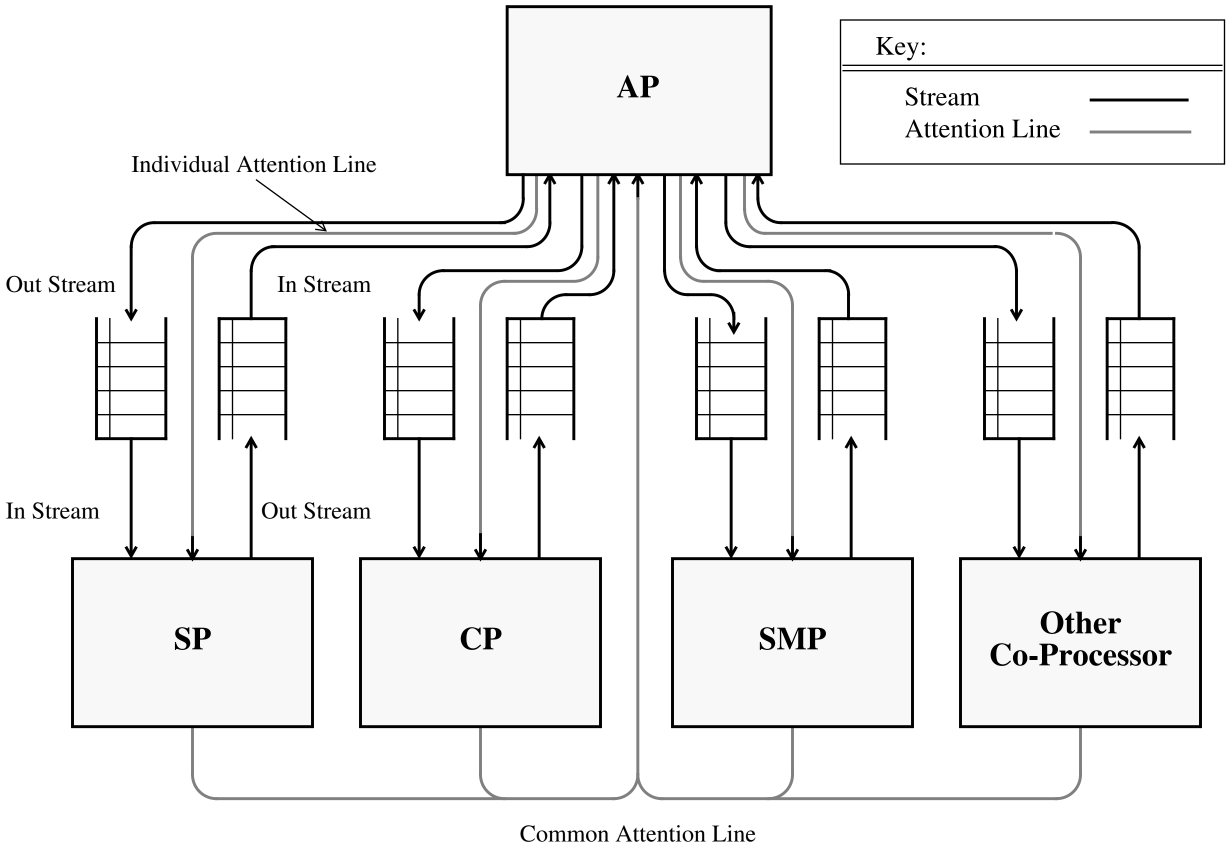


Figure 5: Interprocessor Message Interface (IMI) Model

Multibus interrupt lines are employed to send these interrupts between processors. The AP on each node has an *Interrupt Generator* that can initiate an interrupt on any of four Multibus interrupt lines under program control. An Interrupt Generator is mapped into the Multibus address space for each node so that it may be accessed by any of the processors on that node.

The AP uses a different dedicated Multibus interrupt line to command the attention each of the co-processors, while all of the co-processors use a single Multibus interrupt line to signal the AP. This means that the AP must examine all of the IMI channels when it receives an interrupt in order to find the source of the interrupt, whereas each co-processor assumes that it is receiving an IMI message from the AP. (If there were more interrupt lines available, dedicated lines would have been used to signal the AP as well.) The hardware involved in providing this facility is described in [Northcutt 88c].

The IMI facility is intended to be general-purpose. It supports both synchronous and asynchronous IMI messages, where a synchronous message is one for which the originator of the message "busy waits" until a response (typically, an "answer," that is, a response that includes some data, not just an "acknowledgment") is received, while an asynchronous message is one that is simply sent by the originator without any explicit notion of a response. Following the transmission of an asynchronous message, the IMI channel is once again available for other messages in either direction.

The IMI facility also allows the originator of a message to decide whether or not an interrupt should be sent to the message receiver for each message sent over the channel.

In practice, the full generality of the IMI facility has not been used. To date, interrupts are always used, and synchronous messages are never used. However, this is not because there are no messages that seem to be synchronous in nature. Rather, it has proven to be difficult to use a single IMI channel to handle both synchronous and asynchronous messages simultaneously. In general, the processor on one end of an IMI channel cannot know what the processor on the other end of the channel will next send over the channel. As a result, when a synchronous message is sent over the channel, the originator cannot be sure that the next message received on the channel's in stream will be the response to the synchronous message. Instead, it may easily be an asynchronous message that is unrelated to the synchronous message just sent.

Due to the high volume of data that is typically included in operation invocations, the AP and CP interact by means of shared memory, as well as interprocessor interrupts. In fact, the AP/CP interface is similar to the hardware-implemented *mailbox* schemes used for interprocessor communication in various systems. The interface allows high-bandwidth communication through the use of shared memory, while attaining low overhead and high responsiveness through the use of interrupt-based signaling. This interface also provides for increased communication performance, deriving from the ability of the Application and Communication Processors to effectively move memory through the manipulation of memory mapping tables, rather than performing the costly copying of memory blocks that is normally associated with interprocessor communication activities.

The IMI interconnection architecture has influenced the interface presented by each of the subsystems to the AP in at least two major ways. First, all of the IMI channels connect the AP to some co-processor. There are no IMI channels that connect co-processors directly. As a result, the AP must occasionally act as an intermediary. For example, when a remote operation invocation is performed, the AP queries the Scheduling Co-Processor (SP) to acquire the current scheduling parameters for the invoking thread before sending an IMI message to the Communication Co-Processor (CP) indicating that it should perform the invocation. If the CP and SP were connected by an IMI channel, the scheduling parameters could be acquired by the CP without the involvement of the AP. Since off-loading non-application processing tasks from the AP is one of the goals of Alpha, a richer interprocessor communication architecture would seem to be advantageous.

The fact that the IMI channels act as mailboxes is the second factor that has affected subsystem interfaces. In general, data is sent among processors through the IMI channels. While Alpha has attempted to use shared memory where possible to reduce the amount of data that must be copied among processors, without a true multiprocessor node architecture—where each processor can potentially access all of the memory on the node—this effort cannot succeed completely. For instance, the Kernel Proper on the AP maintains a list of all of the objects and threads that are local to that node. The CP needs to have an identical list to recognize those operation invocations that are broadcast over the Ethernet to which it must respond. However, putting this information in the shared Multibus memory would be impractical—it would take longer to access from the Kernel Proper and it would be severely restricted in size. As a result, in the current implementation two separate lists must be maintained. This increases the amount and type of information that must be explicitly passed from the AP to the CP, thereby altering the interface that would otherwise exist between the AP and CP. (Similar examples can be offered for the AP/SP interface as well.)

3 Kernel Proper

As defined in previous chapter, the Kernel Proper is the portion of the kernel layer of the Alpha operating system that coexists on the Application Processors with both higher-level-system- and user-defined code. The Kernel Proper is at the center of the operating system, surrounded by the kernel subsystems which comprise the remainder of the kernel layer.

The following sections describe the design and implementation of the major functions of the Kernel Proper. First descriptions of the functions provided in support of the system's programming model are given, followed by descriptions of the Kernel Proper's major data structures.

3.1 Support for the Major System Abstractions

This section describes the design and implementation of the portions of the Kernel Proper that provide support for the programming model's major abstractions. The intent of the following discussion is to provide an understanding of the implementation techniques used in Alpha. Throughout this chapter it is assumed that the reader is familiar with the Alpha programming model (as defined in [Northcutt 88b]), and so only design and implementation issues will be discussed.

Specific details relating to the object management, thread management, operation invocation, concurrency control, and access control abstractions are presented here.

3.1.1 Objects

In accordance with the abstract definition of Alpha's objects, objects are simply regions of memory that contain code and data. The data portion of an object consists of several different types of data, with the major division being between statically and dynamically allocated data. An object is created with a unit of static data that is global to all threads that might be executing within the object at any point in time. The size of this data region cannot be increased or decreased, and the static data region consists of both initialized and uninitialized data. The dynamically allocated portion of an object's data is known as the object's *heap* region and can be increased and decreased in size at run-time.

Objects are not permanently associated with any particular context, but rather "float" around in the system's memory until they have operations invoked on them by threads. For this reason, an object has associated with it all of the virtual memory and secondary storage structures necessary for the management of the object. While the data region of an object may be read or written by any thread that has the object mapped into its context, the code portion of an object's memory space is protected in an *execute-only* fashion.

When an invocation is made on an object, the memory that makes up that object is mapped into the invoking thread's context. This is done by manipulating the virtual memory management address maps associated with the invoking thread. The act of mapping an object into a thread's context can be viewed as a "partial context swap," because a portion of the executing thread's context is replaced. However, this activity is considerably simpler (and hence quicker) than a full context swap, because the executing thread does not give up the processor and so the machine state does not have to be saved and restored.

An object can be simultaneously mapped into multiple threads' address spaces on a given node. This permits for concurrent execution of threads within (either the same or different operations of) objects. All threads executing within an object share the object's common (execute-only) code section, as well as the different (read/write) parts of the object's data section.

To help manage the synchronization of threads executing within objects, object control blocks contain references to all of the threads that are currently active within an object.

Furthermore, objects provide protected entry points into their code section by way of a software version of hardware "gates" (i.e., capabilities). This is done by way of a simple, hardware-protected indirection table known as an *object entry point table*, which is generated by the object-language pre-processor [Shipman 88]. The object entry point table 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 operation invocation indices and to locate entry points in the objects. Thus, an object control block also contains all of the permanent capabilities that belong to an object at a given point in time.

More detailed descriptions of the memory management and protection aspects of objects are given later in this chapter.

The preceding discussion focussed on the typical type of object found in Alpha—i.e., a *client object*, an object that exists in the machine's user space. There is another type of object in Alpha that is implemented quite differently. These are known as *kernel objects*—they are actually just kernel extensions appearing to the programmer as standard objects. Kernel objects exist within the Kernel Proper's address space, and do not make use of the system's virtual memory features. While the programmer views the kernel object optimization as just another compile-time-specified attribute of an object, the implementation of kernel objects is quite different from that of client objects. Both kinds of object type specifications and definitions are written in the same manner, 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 control information (i.e., virtual memory information, secondary storage information, capability lists, etc.). With client objects the code and data components exist in separate virtual memory extents, whereas with kernel objects, the code and data of all kernel objects are all combined within the code and data portions 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 control blocks do not contain references to virtual memory management information, or the thread context(s) the object is currently mapped into. The kernel object's control block is a quite simple structure that consists mainly of a pointer to the object's entry point block in the kernel's data region, the kernel object's capability list, and its synchronization data structures.

Just as with client objects, the object language pre-processor generates an entry point table for each kernel object. However, entry point tables for client objects are placed at the start of each object's data region, while the entry point tables for kernel objects are placed in various locations in the kernel's data part.

In addition, kernel objects are linked differently than client objects (i.e., kernel objects are linked with the kernel, as opposed to being separate, stand-alone entities), and 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 (executing in user mode) to gain access into the kernel on invocations is not used. Instead, the pre-processor only generates the code to marshal the necessary parameters and then to make a call to the kernel's operation invocation routine.

3.1.2 Threads

There are two varieties of threads in Alpha—*kernel* threads and *client* threads. Kernel threads exist as both kernel optimizations and fundamental units out of which client threads are constructed. In Alpha, a client thread is bound to its own virtual address space, separate from all others. Kernel threads do not have their own virtual address spaces, but rather execute entirely within the kernel; kernel threads are a simplified form of a thread that exists within the kernel's portion of a virtual address space.

A kernel thread is designed to contain all of the basic (non-virtual memory) information necessary for managing a thread. Kernel threads keep the processor's registers when a thread is de-scheduled (i.e., unbound from the processor on which it is executing), along with copies of the thread's user and supervisor stack pointers. Furthermore, kernel threads contain the state information known as the thread's *environment*, that consists of the information necessary for managing threads within nested atomic transactions, and the information needed by the scheduling facility to perform the desired thread scheduling functions. Kernel threads are implemented in a fashion very much like that of standard processes in more traditional systems.

Each client thread in the system is constructed from a kernel thread—i.e., a client thread consists of a kernel thread with additional information. Client threads differ from kernel threads largely in as much as they include virtual memory information to support the virtual address space that is associated with client threads. A client thread contains all the necessary virtual memory information for the thread's address space, and a reference to the currently mapped client object (i.e., the object in which the thread is currently executing). When executing within the kernel, each client thread uses its own kernel stack. This allows each thread to block on its own stack, saving any state necessary, prior to activating another thread. This serves to simplify the kernel's implementation since otherwise, a communal stack would have to be shared among all threads.

Included in the information associated with a thread is an indication of its execution state—i.e., the mode it is currently in with respect to the scheduling facility. Figure 6 provides the state/transition illustration of thread execution states in Alpha. In this figure, the various states that a thread can be in are represented by circles. The legal transitions between thread states are represented by arcs (which are labeled with the names of the scheduling routines that are used to cause the indicated transitions to occur).

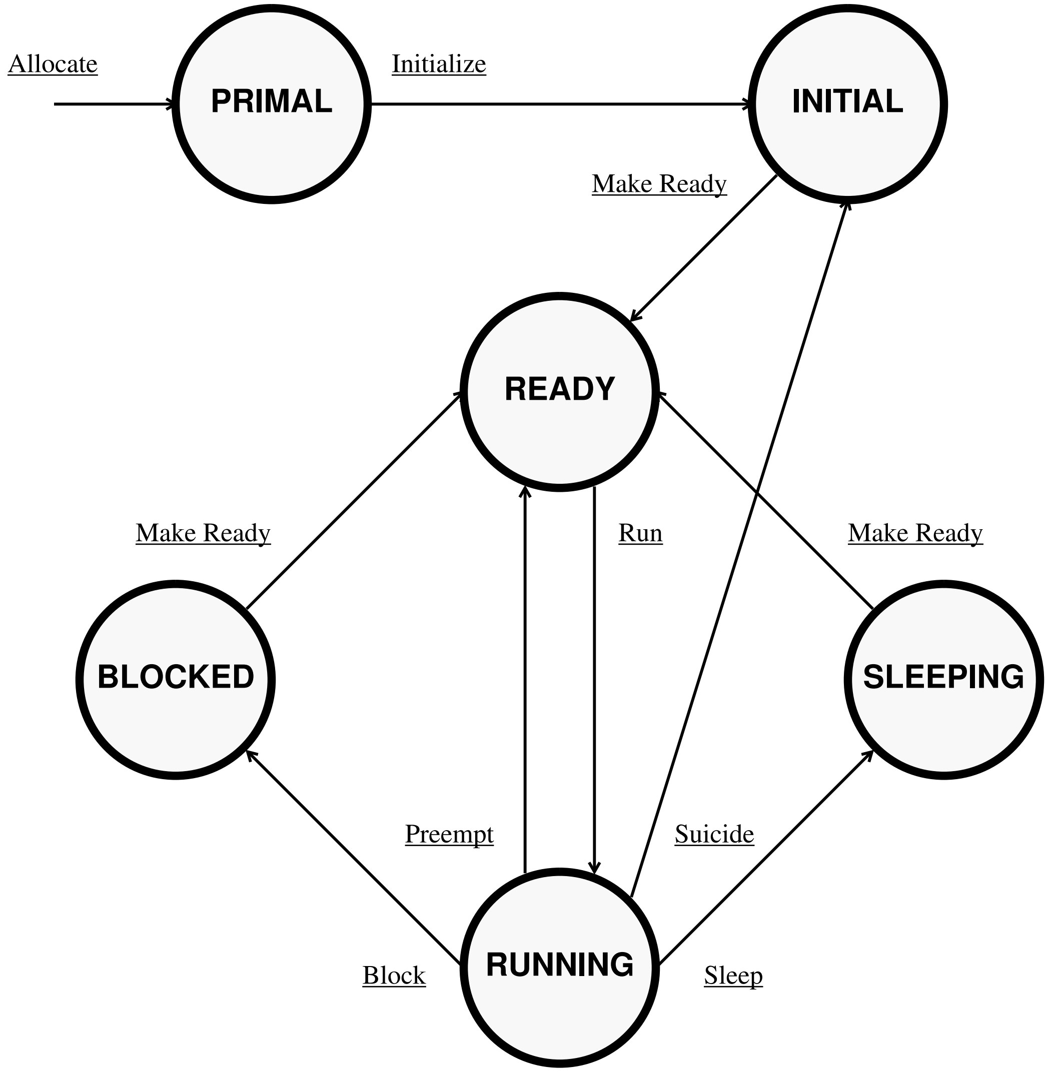


Figure 6: Thread Execution States

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 the invocation. 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.

While each client thread is associated with its own separate virtual address space, it does not occupy the full range of virtual addresses defined for its space. The kernel is mapped into a thread's address space, and space is reserved for objects to be mapped into the thread's address space on invocation. The portion of a thread's virtual address space that is occupied by the thread is made up of three major components—i.e., the kernel stack, client stack, and invocation parameter portions. In the current implementation, a thread must be bound to a hardware supported address space (i.e., *context*) in order for the thread to execute. Furthermore, guard pages are placed between each of the major portions of a thread in order to provide a measure of protection from errant pointers and stack over-/underflows.

The thread switches to execute on its kernel stack when it enters the kernel, and the remainder of the time, it executes on its client stack. The thread's client stack is used for the return addresses and local variables associated with the routines that make up an invoked object's operations. The client stack is automatically extended and contracted as the thread's needs changed in the course of its execution.

In Alpha, both capabilities and (simple or structured) data is passed as parameters between objects on operation invocations. In the current implementation, all invocation parameters are placed in the invocation parameter portion of the thread's address space. The invocation parameter passing mechanism takes advantage of the testbed memory management hardware's ability to perform low-cost transfers of physical memory pages among address spaces. The ability to quickly and easily remap physical pages between contexts 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 operation invocation, the kernel must also move parameter blocks between the invoking and invoked objects. In Release 1, a logical stack of invocation parameter pages is maintained within the thread's memory region, with the currently active operation invocation's parameters always on top of the stack. This approach allows the parameter blocks for the currently active invocation to be maintained at a fixed location in a thread's part of an address space, throughout whatever operations the thread may invoke.

The parameters passed into, and removed from, an object as a result of an operation invocation are placed into data structures that exist within virtual memory page boundaries, known as *invocation parameter pages*. Invocation parameter pages are mapped into different locations in a thread's address space in order to effect the passing of parameters among objects on operation invocations. Each thread has associated with it a pair of active invocation parameter pages—i.e., an *incoming* page that is shared by the current object and the object that invoked it, and an *outgoing* page that is shared (serially) by the current object and any objects it invokes.

Each invocation parameter page is composed of two main parts—the *request* part that contains the parameters to be passed into the invoked object, and the *reply* part that contains the parameters that are to be passed back to the invoking object when the invoked operation completes. Invocation parameter pages are managed in an *overlapped window* fashion. When an operation is invoked, the incoming parameter page is mapped out of the invoking 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 mapped into the invoked object's outgoing parameter page location. 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. Figure 7 illustrates the behavior of the thread's invocation pages in the course of a (local) operation invocation.

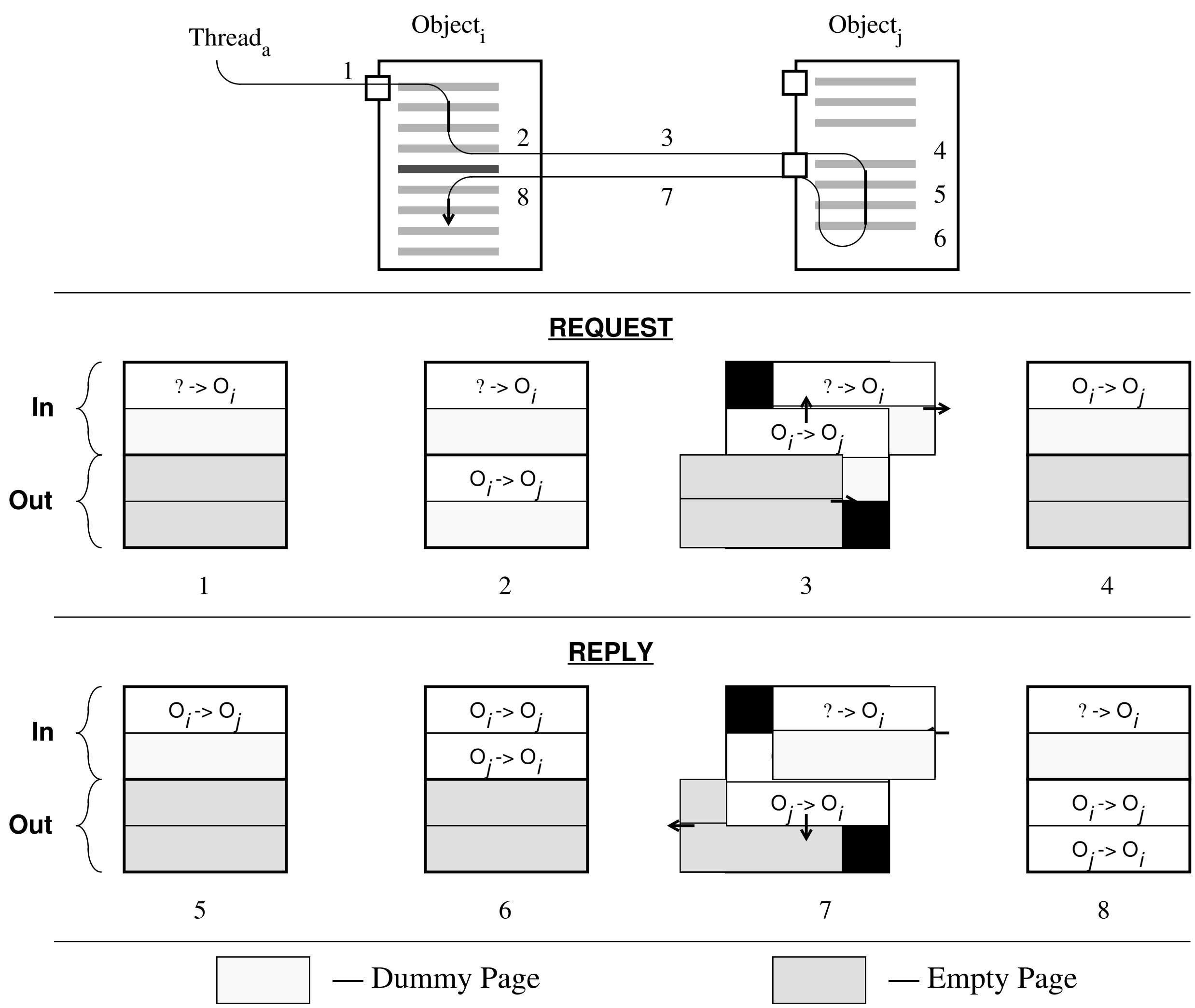


Figure 7: Parameter Page Manipulation on Invocation

3.1.3 Operation Invocations

The operation invocation facility has been implemented to provide a uniform interface to programmers both above and within the kernel. 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 the code needed to perform operation invocations. An invocation begins with the marshaling of parameters, which is 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 Alpha 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 does provide hooks to add such translation routines if the need should arise in the future. If a real object programming language were to be developed for Alpha, 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 thread traps into the kernel. This causes the application processing element to enter supervisor mode and begin the invocation process. The first thing done by the operation invocation trap handler is the determination of the invocation type, as indicated by information encoded in the target object's Alphabit identifier. There are three different invocation types, which depend on the object that is the destination of the invocation—i.e., *system service*, *local*, or *remote* operation invocations. Figure 8 illustrates the logical structure of a node's operation invocation facility, showing the major components of (and paths through) the facility.

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 control blocks). 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.

System service operation invocations are used to obtain the services that are provided by the system to the programmer in the form of system-defined objects. For system service invocations, the normal capability translation and validation steps are simplified. This invocation path represents one of a series of "short-circuit"-style optimizations that are provided to compensate for the performance costs of unifying the system's interface by way of the operation invocation facility.

If the destination of an invocation is not a local system service object, a lookup operation is performed on the local Dictionary (see Subsection 3.3.2) to determine if the destination entity is local to this node. If the destination entity's 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 (described in Chapter 5) 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.)

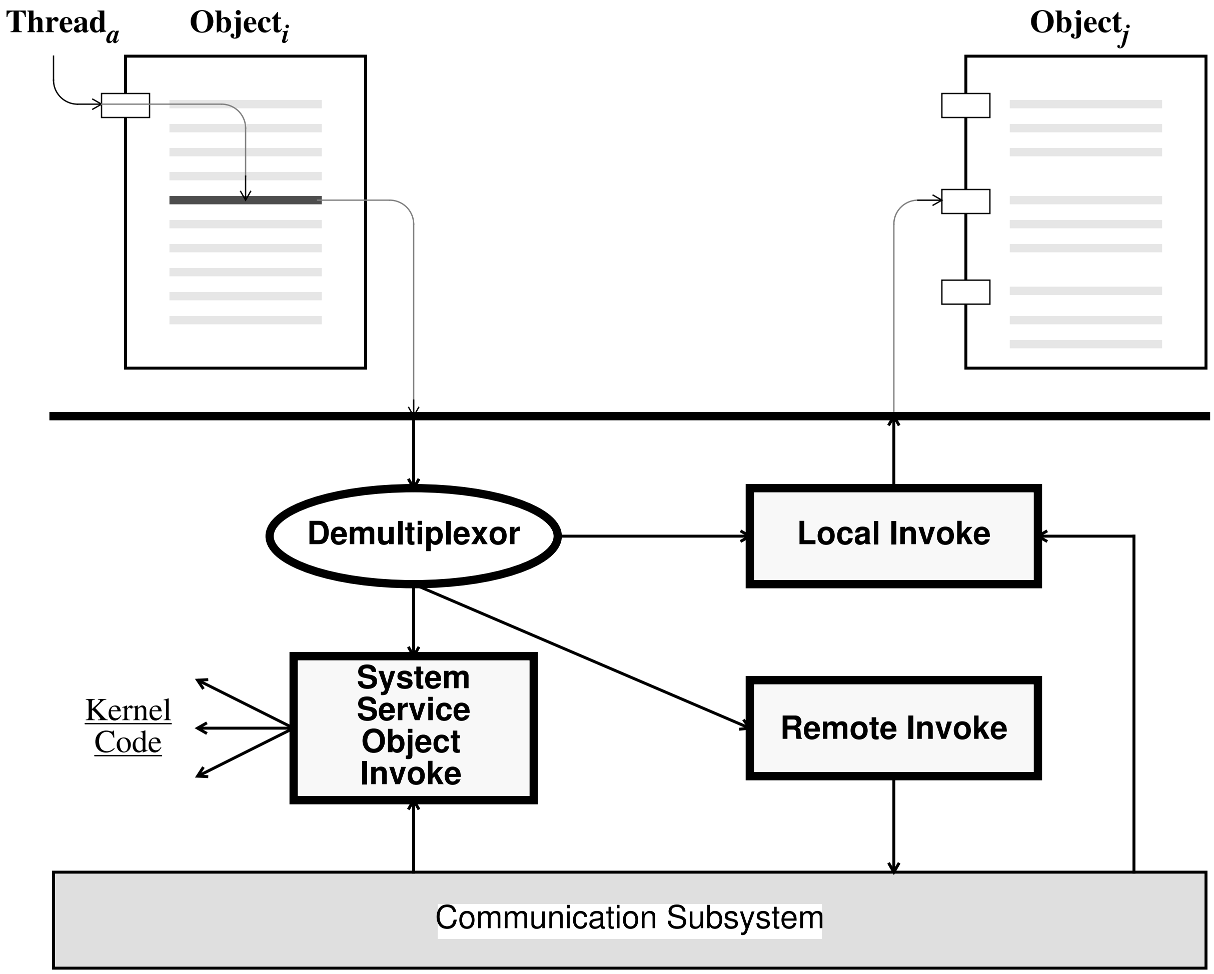


Figure 8: Operation Invocation Facility

In the case of both local and remote operation invocations, the invocation procedure continues with the translation and validation of any capabilities passed as parameters. The use of capabilities is described in detail in Subsection 3.1.4, 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—i.e., the capability used to indicate the object that is the destination of the invocation. If a passed capability is invalid (i.e., it is non-existent or its usage is restricted), the invocation is terminated and a failure indication is returned. When all of the capabilities in an invocation are successfully translated, the invocation proceeds by calling the kernel's internal invocation routine.

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 entities. 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 message, puts it in the Communication Subsystem's mailbox, and signals the Communication Co-Processor. This message contains an opcode indicating that this is to be a remote invocation, a pointer to the passed the invocation parameter page, the identifiers of the invoking object and thread, and the invoking thread's execution environment. Once the remote invocation routine passes its 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 message from the Communication Subsystem, it locates the invocation 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 then 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 Subsection 3.2.1). 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 9 illustrates the different paths that can be taken through the operation invocation facility.

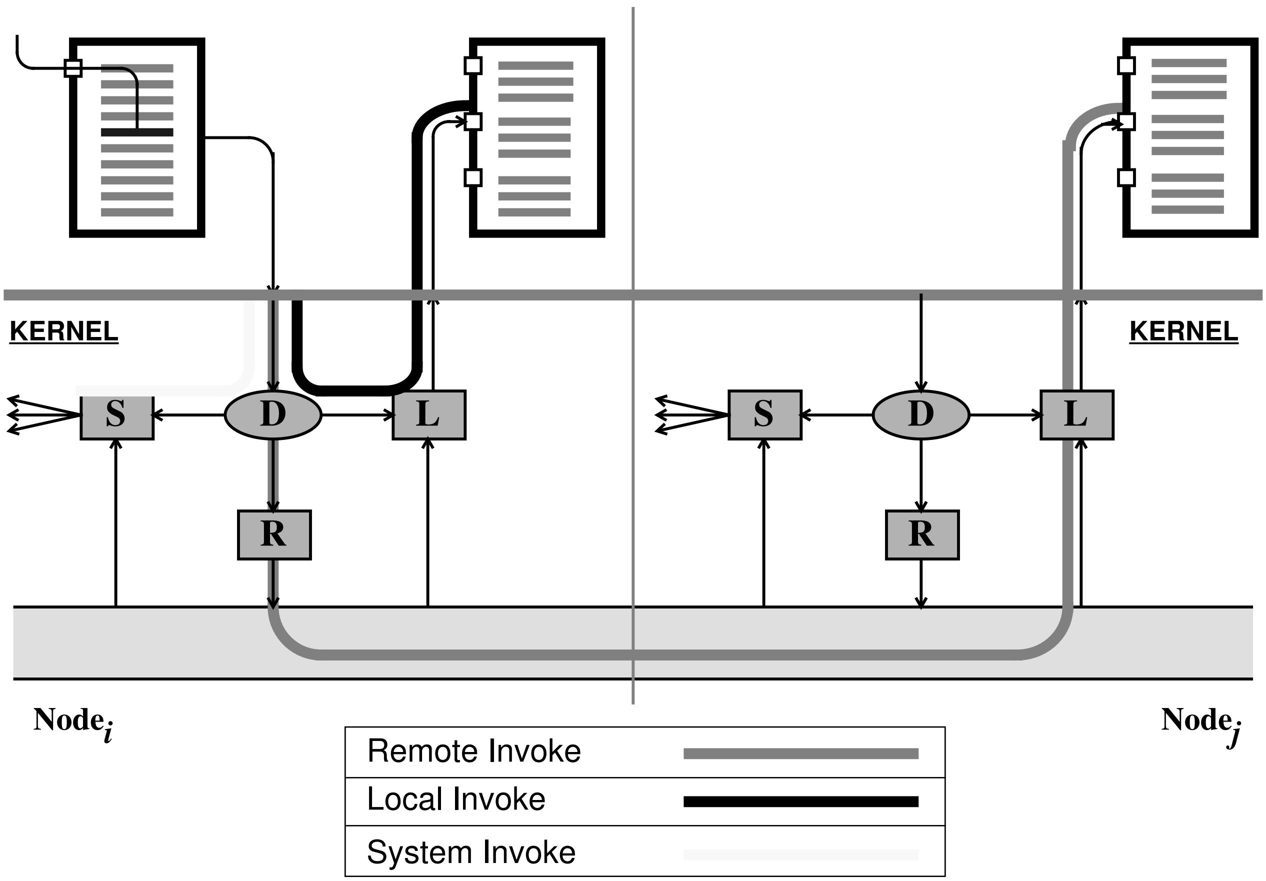


Figure 9: Types of Operation Invocation

3.1.4 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 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 capability list (*c-list*) consists of an array of capability descriptors, located in each object's control block, that are 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., 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.

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 cannot 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.

3.1.5 Thread Synchronization

The Alpha kernel provides a set of kernel-defined objects that support the kernel's concurrency control mechanisms—i.e., *semaphore* objects and *lock* objects. Each of these mechanisms are described in turn in the following text.

3.1.5.1 Semaphore Mechanism

The creation of an instance of a ***Semaphore*** object involves the allocation and initialization of a semaphore data structure. Semaphore data structures consist of a variable containing the semaphore's current state, and the list of threads that are currently blocked waiting on the semaphore. The kernel takes advantage of the fact that capabilities for semaphores cannot be passed to place the semaphore data structures within an object's control block. It similarly takes advantage of the small size of semaphore data structures, and statically allocates a fixed number of semaphore data structures for each object.

When a new instance of a semaphore is created, an unused semaphore data structure is located, initialized with the count value given as a parameter to the create operation, and then a capability is returned to the invoking object. A create operation on the ***SemaphoreManager*** object returns a failure indication if a semaphore data structure cannot be allocated.

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 control block, but a special type of capability can be used. To the client, 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 usual unique identifier and restrictions vector. Rather, it contains an indication of the special nature of the capability and an index into the object's semaphore array for the desired semaphore.

A semaphore data structure contains a count field (that is manipulated by the operations defined on semaphores), a list of the threads that are blocked on the semaphore (waiting for the count to become greater than zero), and a list of the threads that are currently using the semaphore (i.e., 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 "waiting thread" 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. Work is underway to extend the scheduling subsystem to include the management of blocked threads as part of its functionality.

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 encounter an exception (e.g., an aborted atomic transaction, a missed hard-deadline, or as part of the thread repair function) while holding a semaphore. In the course of cleaning up objects following thread exceptions, the kernel checks the list of threads associated with each of the affected objects' semaphores, and for each occurrence of the thread being aborted in a semaphore's list, the kernel may choose to issue a `V` operation on that semaphore (or not, depending on the mode in which the semaphore is to be used).

3.1.5.2 Lock Mechanism

Lock objects are similar to ***Semaphore*** objects in their 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 control block.

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.

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 waiting thread 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 semaphore data structures, those of locks 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.

3.2 Ancillary System Functions

In addition to providing support for the basic abstractions in Alpha, the kernel provides a collections of other services that are needed to round out the functionality of this layer of the system. Included among these functions are features to manage the preallocation of critical system resources, the handling of run-time exceptions, and the initialization of the individual nodes within the system. Each of these functions are defined in the following subsections.

3.2.1 Critical Resource Preallocation

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 kernel 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 control blocks. 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* routines) 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.

3.2.1.1 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 Storage Management 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 Storage Management Subsystem that is responsible for copying the victim page to backing store. 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 Storage Management Subsystem 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 Storage Management 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).

3.2.1.2 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 *skel-*

etons, and consist of generic control blocks 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 control blocks, and in the case of client threads or objects, virtual memory information. 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 virtual memory structures need not be obtained and initialized.

3.2.2 Exception Handling

Any system with an application requirements similar to Alpha's demands some form of exception handling facility. These features (i.e., distribution, real-time, and transactions) result in exceptions that are best manifest in the form of asynchronous diversion of the point of program execution control. In Alpha, this means that a mechanism is needed that will allow the programmer to deal with the exceptions which are a fundamental aspect of the system's nature, in a manner that is compatible with the other aspects of the programming model.

Alpha provides an exception handling facility that handles, in a unified fashion, all of the different types of exceptions that may occur. The Alpha exception handling facility is based on the concept of *exception blocks*—i.e., regions of code marked by begin/end operation invocations. The system automatically places an exception block around each operation, time constraint region, and atomic transaction that a programmer generates. In addition, the programmer may explicitly specify exception blocks that consist of arbitrary regions of code.

Each time an exception block is entered, the kernel saves the machine state and bumps the exception block nesting depth counter (both of which are stored in the thread's control block). When an exception block is exited (normally or otherwise), the kernel discards the thread state that was saved at the beginning of the block. The general intent is that when an exception block is entered, the thread's execution state is saved so that should an exception occur, the thread can be vectored back to the start of the block and shunted off to exception handling code. This mechanism behaves much in the same way as the UNIX "longjump" function—the thread is returned to a given point in the code, with the its complete data and execution state intact.

The thread's exception block state is managed in a stack-oriented fashion due to the fact that exception blocks are entered and exited in a strictly nested manner. Should an exception occur while a thread is executing within an exception block, the kernel signals the exception, and a depth indication is given with it to indicated how far the thread is to roll back through its exception stack. As a thread rolls back, the previous exception state is discarded, and the new state is used to vector the thread to the desired exception handler code. In addition, a thread's scheduling parameters are popped as necessary while it rolls back through its exception blocks. The thread encountering an exception continues unrolling until its head is positioned to begin executing its normal-flow code at the desired exception level.

Threads include their exception nesting depths as part of their environment—it is passed around with the thread as it moves among nodes, and surrogate threads inherit the exception stack depth (but only maintains its own portion of the exception stack).

In Release 1 of Alpha, exceptions are signalled to threads by jamming an exception return frame onto the thread's kernel stack. This is done so that when the thread attempts to resume its execution, it is not allowed to continue, but is instead forced to execute the kernel's exception handling facility code. In effect, this mechanism inserts the kernel's exception code in-line with the target thread's execution, regardless of where or what state the object is executing in when an exception occurs (e.g., executing within an object, performing a software trap, or handling a hardware interrupt).

In order to jam exception frames onto threads' kernel stacks, it is necessary to know where the top of a stack is when the thread enters the kernel. This is easy to determine in the case of client threads because they execute on their client stack until they trap into the kernel. It is more difficult in the case of kernel threads because they have to execute on their kernel stack at all times. To deal with both of these cases, the system was designed so that a thread's kernel stack frame is marked the first time that it enters the kernel—whether via a trap, interrupt, or procedure call—and it is cleared when the thread leaves the kernel. This is accomplished by using an atomic test and set instruction on each entry into the kernel in order to determine if it marks an initial transition into the kernel, and if so the thread's kernel stack frame is marked, otherwise nothing is done.

This exception signalling mechanism is idempotent in its effects, and it can therefore be used to repeatedly (e.g., as a result of multiple, "simultaneous" exceptions), and the kernel's exception facility will ensure that the thread will roll back through its exception blocks until it reaches the proper exception level. Furthermore, an exception can be generated at any node, and all affected thread segments will be notified. The exception facilities on separate nodes precede with their operations simultaneously, with the communication subsystem providing the synchronization necessary to ensure that all segment cleanup is complete before the new head of the thread continues its execution.

3.2.3 Initialization

On power-up, all processing elements in a node initialize their hardware resources, perform self-test operations, and then await a load module to begin executing. Typically, each node receives a load module that contains the code and data needed to bring up a replica of the Alpha kernel.

A node's load module is created by the programming language support tools defined in [Shipman 88], and take the form of the example load module shown in Figure 10. The same load module can be used for each node in the system, with per-node differentiation being based on the unique node identifiers and occurring following the initialization of the generic kernel on a node. It is also possible to create node-specific load modules that are tailored to individual nodes (or sets of nodes).

As shown in Figure 10, the canonical Alpha load module consists of a load module header (that contains a first-level loader program and a description of the sub-modules that follow the header), and a set of load sub-modules destined for the node's processing elements. Each of the sub-modules contains the code and data for the various components of the kernel (i.e., the Kernel Proper, the Scheduling Subsystem, the Communication Subsystem, and the Storage Management Subsystem).

As part of the monitor's power-up sequence, the low-core region of the Application Processor's memory is initialized. This region of memory contains the processor's exception vector table, the monitor's data, and an initial stack page for the kernel. The Alpha load module is loaded into the Application Processor's local primary memory above the defined low-core region, and execution begins in the loader program supplied within the load module. The default first-level loader program distributes all of the sub-modules to their intended co-processing element, and relocates the Kernel Proper, and begins execution of the Kernel Proper.

Once the sub-modules are distributed to the co-processors, the initialization of the Kernel Proper begins. The loader relocates the Kernel Proper so that it starts at the second memory segment, and manipulates the memory mapping hardware so that it appears at this location in all contexts. The code and data regions of the Kernel Proper are followed in memory by all of the "wired" (i.e., memory-resident) kernel objects.

The Kernel Proper's initialization begins by modifying the hardware memory translation tables to construct the desired virtual address mappings for the kernel. The kernel's virtual memory initialization function sets aside the physical pages for low-core and the Kernel Proper's code. 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 can be allocated on demand.

Once this low-level initialization is complete, the Kernel Proper's facilities are initialized and the initialization object is instantiated and the initialization thread begins execution within it. It is up to the initialization thread to instantiate the threads and objects that are to run on the node, based on the information contained in the initialization object.

The co-processing elements in a node follow a power-up initialization sequence similar to that of the Application Processor, and then await the distribution of their sub-load-modules from the Application Processor. The loading of co-processor load modules is accomplished by way of the interprocessor communication facility and the node's shared memory. Each co-processor then begins the execution of the loader program supplied with their load sub-module. In this manner, an entire node is brought up and made ready to begin the execution of application code.

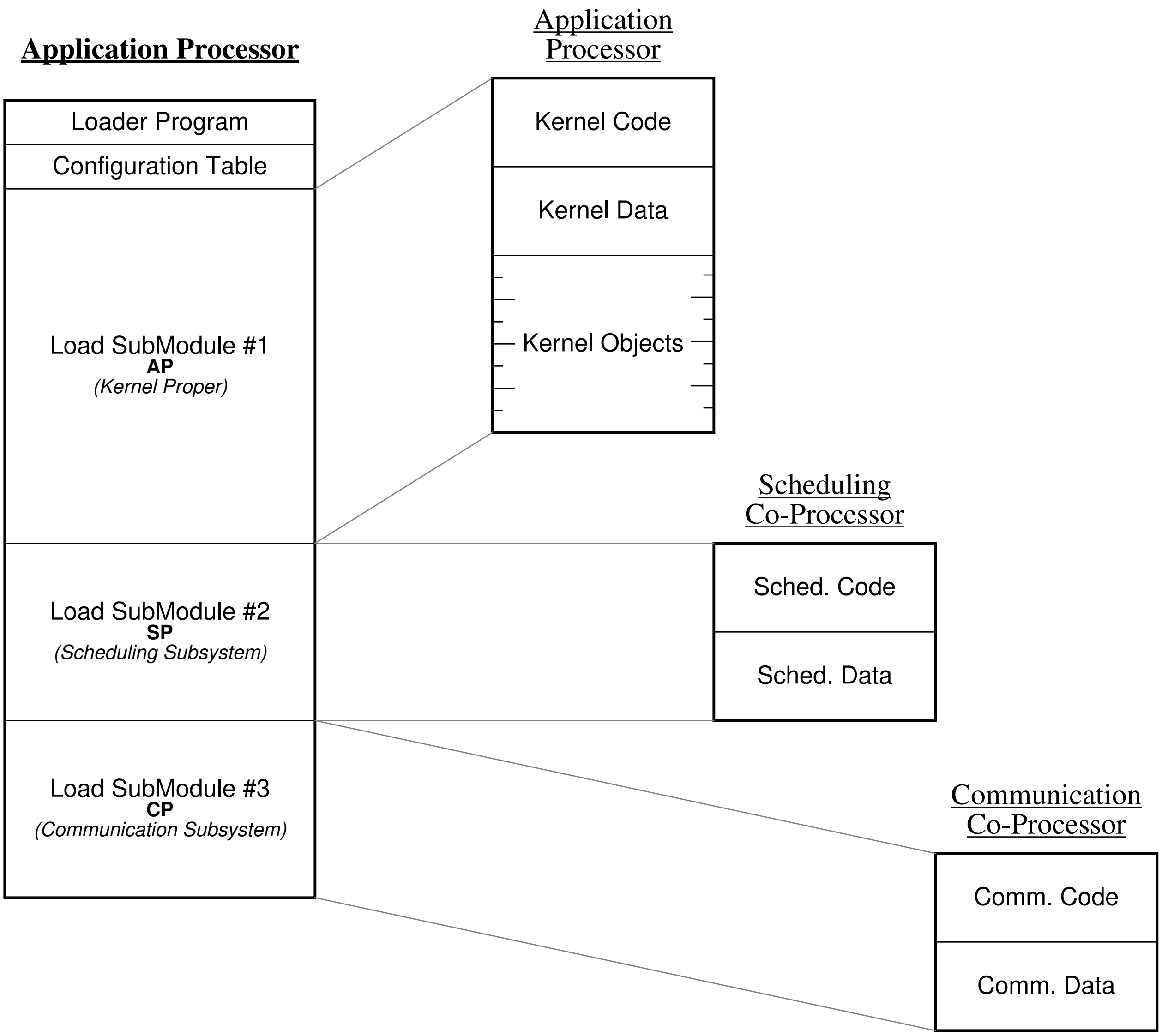


Figure 10: Application Processor Load Module

3.3 Key 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.

The fundamental programming entities in Alpha are represented within the kernel by control blocks, all of which are given globally unique identifiers and collected into a directory data structure to allow individual ones to be located by their identifiers. The purpose and contents of each of these data structures are described in the following subsection. In addition to these, a significant portion of the kernel's data structures have to do with virtual memory management. These structures are also described within this section.

3.3.1 Control Blocks

The kernel provides its clients with an object-oriented programming model to enhance the organization and modularity of applications, and the operation invocation facility provides physical location independent access to objects in the distributed system. The benefits that the client derives from these features of the kernel are equally useful for programmers within the kernel. It is frequently the case within the kernel that references are made to entities that exist at remote nodes (e.g., secondary storage images—see Section 6.2). For this reason, the support for distribution provided by the kernel's operation invocation facility is made directly available to the kernel itself by representing all entities as entities for which remote access is required or may be desirable.

To this end, the kernel maintains a variety of control blocks for each such internal, object-like entity in the system. These data structures are known as *control blocks*—i.e., kernel data structures that exist on the same node with the entities they represent. Each kernel-level entity in the system (e.g., a thread or an object) has a corresponding control block maintained by the system. Any entity represented in the kernel with a control block 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 a client object or the kernel itself.

All Alpha control blocks consist of a common header, followed by fields that are specific to the particular entity that is being represented. The control blocks' common header contains: a globally unique identifier used to access the data structure, an indication of the type of entity being represented, a time-stamp that indicates the 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 control block (that is used for chaining them together within a common hash table entry).

Each control block has a globally unique, non-reusable identifier in its header known as an *Alphabit identifier*. 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 entity 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.

This structure is not visible outside of the kernel, instead they appear to the client as unique, uninterpreted bit strings. 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.

The following paragraphs provide a description of each of the different types of control blocks defined in the current release of the Alpha kernel.

3.3.1.1 Kernel Control Block (KCB)

This data structure represents the portions of the kernel that use the virtual memory facility. This control block is used in order to make the kernel appear similar to client objects with respect to its interactions with the virtual memory and secondary storage facilities. 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 kernel's virtual memory data structures is provided in Subsection 3.3.3.

3.3.1.2 Client Object Type Control Block (COTCB)

This data structure contains the kernel's representation of an object type definition. This structure contains a standard control block header, a print string name for the type description, a flag that indicates if the type is elaborated (i.e., whether an image of it currently exists in primary memory), a file control identifier, and a c-list for all of the well-known capabilities for this type of object.

3.3.1.3 Client Object Control Block (COCB)

This data structure maintains all of the control and status information for a client object. It includes the standard control block header, all of 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.

3.3.1.4 Client Thread Control Block (CTCB)

This data structure represents a client thread and includes all of the virtual memory information for the thread, a pointer to the thread's client stack, a pointer to the control block for the object in which this thread is currently active, and a pointer to the control block for the kernel thread associated with this client thread.

Each CTCB has an associated Kernel Thread Control Block (KTCB) which contains the bulk of the information about the thread. All threads begin as kernel threads, and are represented with by KTCBs. When a thread (dynamically) transitions from being a kernel thread to being a client thread, a CTCB is created and attached to the original KTCB. Likewise, if a client thread returns to being a kernel thread, its CTCB is removed and discarded.

3.3.1.5 Kernel Object Control Block (KOCB)

This data structure maintains all of the control and status information for a kernel object. It includes the standard control block header, 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.

3.3.1.6 Kernel Thread Control Block (KTCB)

This data structure provides the internal representation of a kernel thread. In addition to the common header, this control block consists of the thread's identifier, a pointer to its kernel stack, the thread's exception stack, a (mutually exclusive) pair of pointers to either a client thread control block or a kernel object control block, and the thread's *environment*.

In the current implementation, a 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 with respect to preemption (i.e., whether it is deferred or not), the current number of invocations to roll-back on abort, the thread's global importance, the thread's current time constraint, the shape of the current time constraint's time-value curve, the expected amount of computation time required for the current time constraint block, the amount of time that has elapsed since the start of the current time constraint block, and the amount of processing time that has been accumulated by the computation in its current time constraint.

3.3.1.7 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). It supports an operation invocation interface between the virtual memory facility and the Secondary Storage Subsystem. These control blocks 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 control block 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 control block within secondary storage.

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

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 control block. 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).

3.3.2 The Dictionary

The *Dictionary* is a global data structure that contains pointers to all of the control blocks 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 control blocks. A local partition of the Dictionary is implemented as a hash table of pointers to local control blocks, and accessed by Alphabit identifier. As control blocks are instantiated, pointers to them are entered in the local Dictionaries. Once a control block has been found in the Dictionary, a pointer to the desired data structure is returned so that the associated data structures can be manipulated at will.

In support of the form of logical network addressing used in the current release of Alpha, the Communications Subsystem has shared (read-only) access to the Dictionary.

Figure 11 illustrates a local portion of the kernel's Dictionary with instance of various types of control blocks entered into it, and being indexed into by an Alphabit identifier.

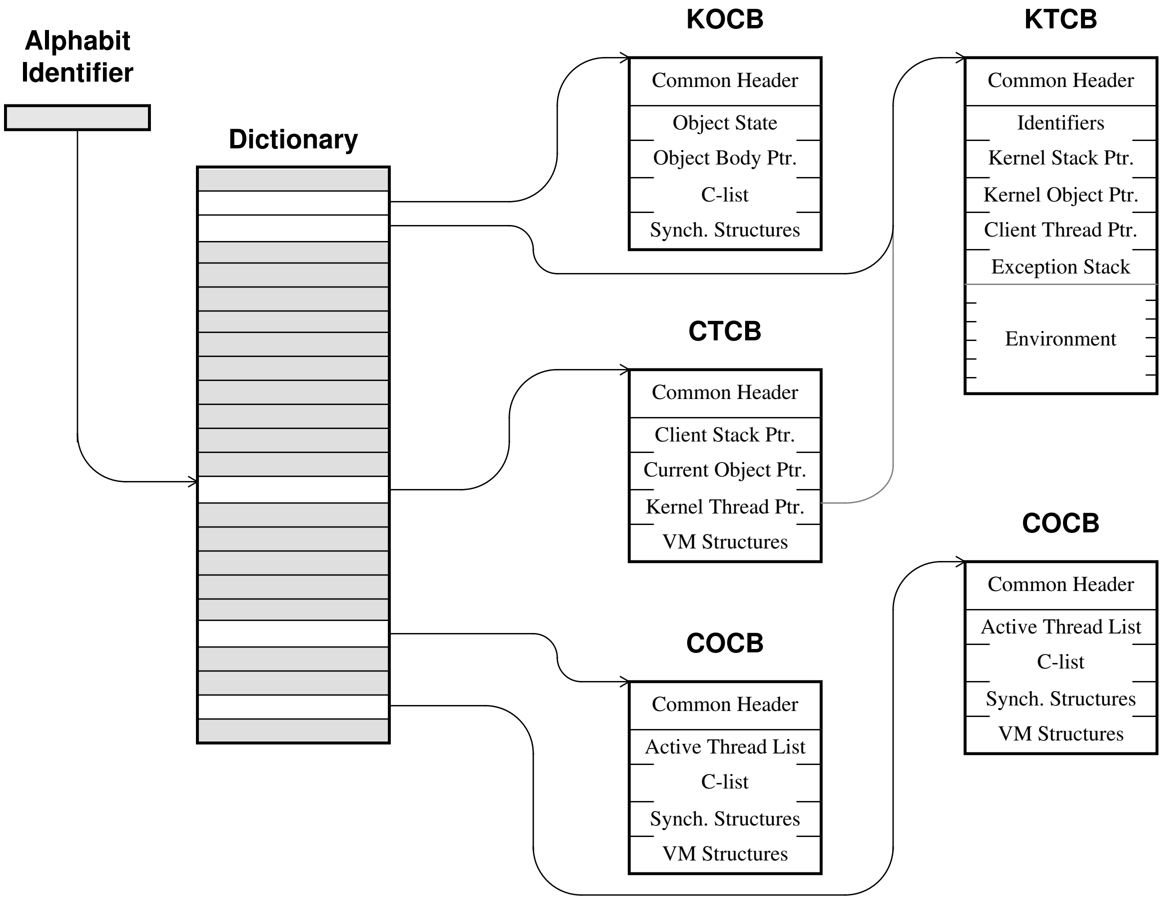


Figure 11: Alpha Dictionary and Control Blocks

3.3.3 Virtual Memory Structures

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. For purposes of modularity and hardware-independence, an effort was made to separate these components in the implementation of Alpha.

This subsection only describes the data structures associated with the system's virtual memory management. The functionality of the virtual memory facility is described in Chapter 7.

Before the virtual memory data structures can be described, it is necessary to be familiar with the memory management hardware found in the Alpha testbed. Following a description of the structures associated with the current testbed's memory management hardware, a description of the additional data structures used to manage entire virtual address spaces (i.e., *contexts*) is given. Finally, the data structures used in the management of individual units of primary memory (i.e., *memory pages)* are described.

3.3.3.1 Memory Management Hardware

Details concerning the Alpha testbed hardware are given in [Northcutt 88c], however the most significant aspects of the Application Processor's memory management structures are summarized here.

The Sun-2 processor board used as the Application Processor in Release 1 of Alpha consists of a MC68010 CPU that issues 24 bit virtual addresses, along with a custom designed Memory Management Unit (MMU) that translates virtual addresses into 24 bit physical addresses and provides a form of access protection.

The structure of the Sun Microsystems version 2.0 MMU is shown in Figure 12. Conceptually, this MMU consists of a two level of indirection look-up table, while at the next level of detail down it consists of:

- a *Context Register* that contains an index into the first of the two translation tables. This index indicates the current address space (i.e., context) that the processor is executing within by referring to a contiguous set of memory segment descriptors.
- a *Segment Map Cache* (SMC) that contains a set of descriptors for a set of memory *segments*—i.e., contiguous sets of memory pages. Each segment descriptor contains an index into the next-level translation table.
- a *Page Map Cache* (PMC) that contains a set of descriptors for a set of memory *pages*—i.e., fixed-length, contiguous spans of memory addresses. 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.

This MMU hardware was designed to provide simple, high-performance virtual address translation, and can be managed in such a fashion as to perform like an address translation cache. Switching between contexts that are loaded in the MMU is an extremely efficient operation (i.e., simply writing to the context register), however only a fixed number of contexts can be loaded into the MMU at a time. Therefore, groups of entries in the MMU tables must be multiplexed among the set of contexts (i.e., threads, in Alpha) that are currently active in the system. Furthermore, all memory protection is concentrated at the page level, and so entries in the SMC do not include any information beyond an index into the PMC. Therefore, a group of entries in the PMC must be reserved for reference by invalid segments (i.e., segments with no pages associated with them).

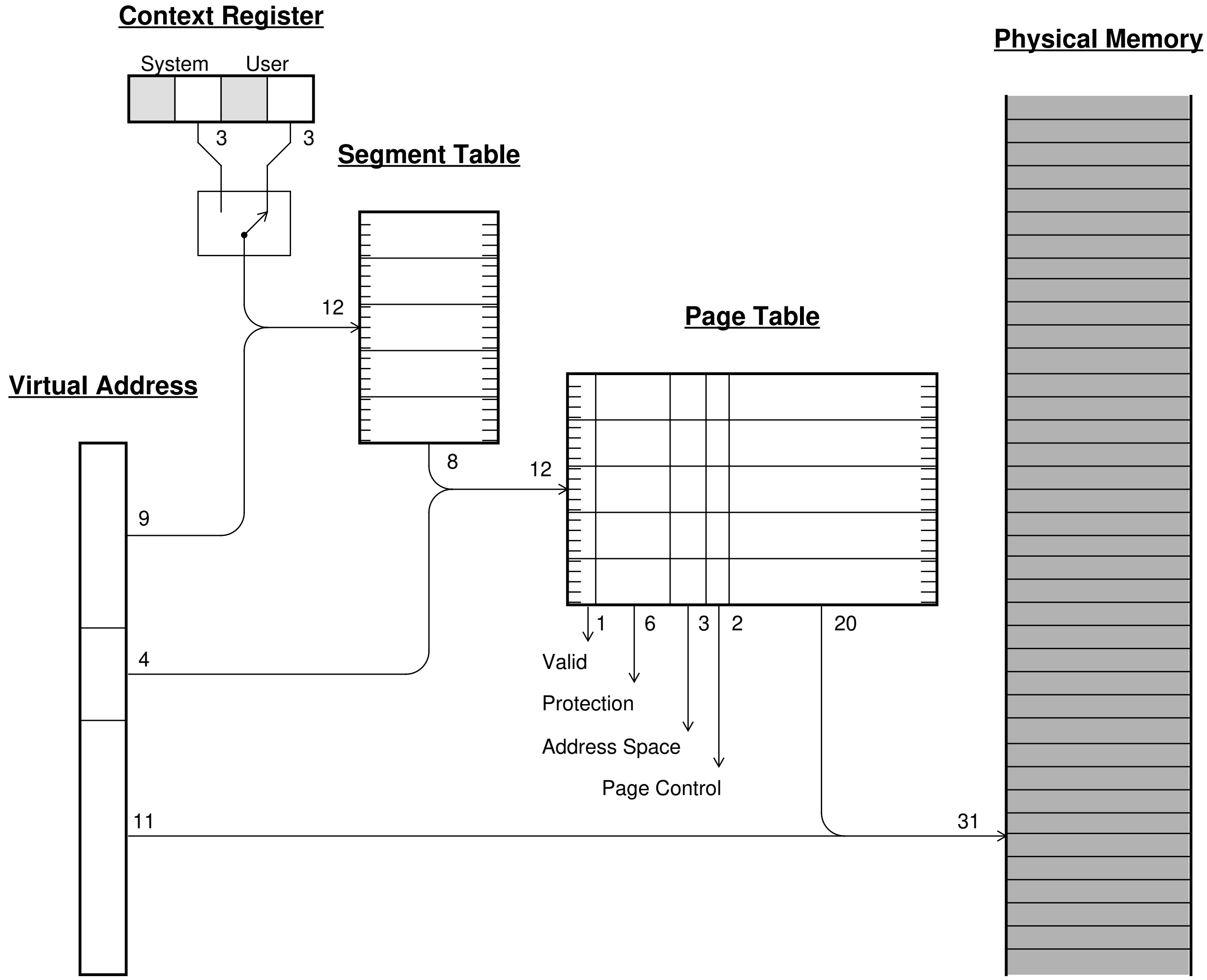


Figure 12: Memory Management Hardware

It should be noted that the Sun-2 MMU provides support for standard, process-oriented programming models (and UNIX in particular), and does not provide any special support for the thread/object model of Alpha. The Alpha programming model might be better supported by an MMU structure similar to that found in the Digital Equipment Corporation's VAX processor line. The VAX MMU manages the caching of translation tables in hardware (via a Translation Lookaside Buffer), and supports a form of partial context swapping (via the SYS, P0, and P1 divisions of contexts) that is compatible with the implementation of operation invocations in Alpha.

3.3.3.2 Context Management Structures

In Alpha, the canonical virtual memory context contains the Kernel Proper, a client thread and a client object. Each of the entities that appear in a context are represented by control blocks within the kernel—i.e., the kernel control block (KCB), a client thread control block (CTCB), and a client object control block (COCB). Each of these control blocks contains a field known as the *segment map image*, that contains the segment descriptors for the portion of a context that the associated entity occupies. 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 control blocks.

In addition to the segment map images, these control blocks contain a collection of pointers to the data structures that represent the functionally related collections of memory segments (i.e., *Extents*) that make up an address space. These data structures are known as *Extent Descriptors*, and consist of:

- a pointer to the control block to which the described 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 control block associated with the Extent,
- the starting and ending virtual addresses of this Extent, relative to its context,
- a pointer to the next Extent belonging to the given control block (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.

Memory Extents and their use in Alpha is discussed further in Chapter 7, while Figure 13 illustrates how they are used in conjunction with other virtual memory data structures.

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 control blocks referring to a particular Extent are chained together, and a reference count is maintained to determine the number of control blocks that are currently making use of the Extent. The Extent Descriptor also serves to link together the virtual memory data structures of a control block (i.e., the control block itself, its Extent Descriptors, and their segment-level data structures) and the secondary storage images associated with its Extents.

3.3.3.3 Page Management Structures

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 control blocks 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.

3.3.3.4 Auxiliary Memory Management Structures

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 control block and the client object control block 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 *in-use* 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 13 provides an illustration of each of the virtual memory data structures used in Alpha.

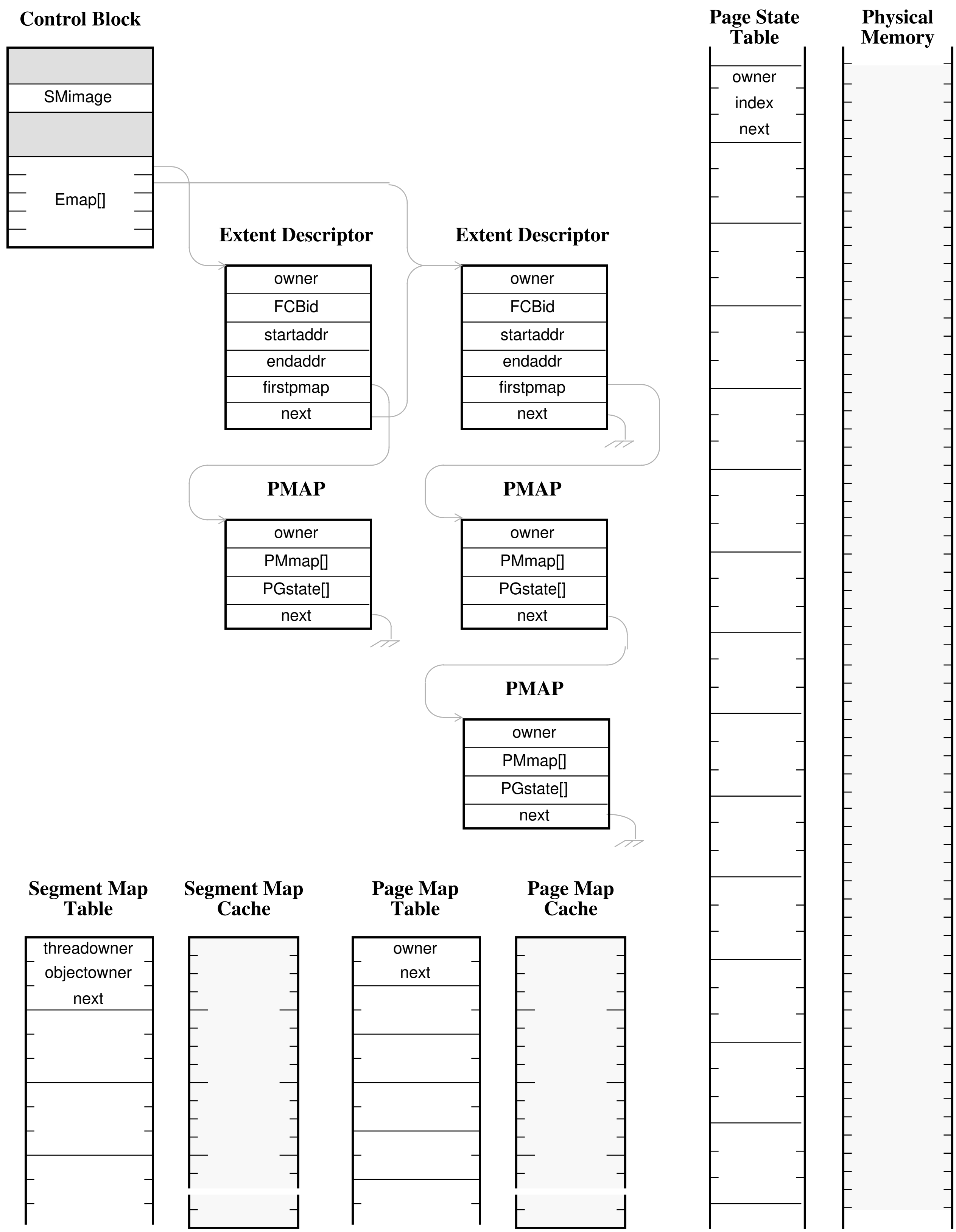


Figure 13: Virtual Memory Data Structures

4 Scheduling Subsystem

The Scheduling Subsystem is one of the principle components of the Alpha kernel, and is related to the other components of Alpha in the manner shown in Figure 2. In the initial release of Alpha, the Scheduling Facility consists of two parts—one that resides in the Kernel Proper and executes on the Application Processor, and another which is known as the Scheduling Subsystem and executes on the Scheduling Co-Processor. These two components execute concurrently and interact via the Interprocessor Message Interface (IMI). The Scheduling Subsystem is different from traditional scheduling facilities in a number of ways—e.g., the Alpha Scheduling Subsystem:

- is cleanly and completely partitioned from the remainder of the kernel by way of a well-defined scheduler interface (enhancing both its potential for parallelism and its modularity),
- is able to operate in parallel with the execution of the application code and Kernel Proper (allowing the execution of computationally demanding scheduling algorithms that would not be practical if the scheduling function were performed in a serial fashion),
- is capable of performing autonomous (i.e., unsolicited) operations on the Kernel Proper (e.g., forcing a context swap—i.e., the suspension of the currently executing thread and the continuation of execution with a different thread),

In the first release of Alpha, the Application Processor is somewhat of a slave, with the Scheduling Subsystem as its master—i.e., the Kernel Proper executes the idle thread and awaits dispatch commands from the Scheduling Subsystem. The Application Processor executes the given thread until another dispatch command arrives from the Scheduling Subsystem, or until the executing thread gives up the processor (i.e., blocks). The Kernel Proper informs the Scheduling Subsystem of all scheduling events which occur on the Application Processor (e.g., when a thread blocks, does a remote invocation, releases a logical synchronization token, or modifies its time constraint attributes). In this configuration, the Kernel Proper is able to dedicate the bulk of the Application Processor's cycles to the execution of application tasks, while the Scheduling Subsystem concurrently carries out the computations necessary to determine how best to manage the Application Processor's cycles.

The following subsections provide the rationale for the Scheduling Subsystem's design, an overview of how the kernel-level programmer makes use of the features provided by the scheduler, a description of the Scheduling Facility's implementation, a discussion concerning the definition of scheduling policies (including an example policy module), and a brief overview of the yet to be completed work on the scheduler.

4.1 Scheduler Requirements

All Application Processor cycles in an Alpha node are intended to be managed by the node's Scheduling Subsystem. The intent here is that the Application Processor execute application code continuously, with the amount of cycles expended on the execution of system code kept to a minimum.

In such a scenario, the scheduler makes all of the decisions concerning the management of Application Processor cycles. In general, resource management decisions must be made whenever there is contention for a resource. In the case of multiprogramming system, there is almost continuously some form of contention for processor cycles. One form of contention stems from the fact that such a system, by its very nature, multiplexes the Application Processor(s) among multiple software activities. This type of contention persists as long as there is more than one computational activity in the system (which is almost always the case). Another form of contention arises when there are insufficient cycles available to execute the current set of computational activities within their given time constraints. Whereas all systems must deal with the former type of contention, real-time systems must be particularly concerned with the latter type. A scheduler in a real-time system should resolve the first type of contention in such a way as to minimize the second form of contention, and when the latter form of contention occurs the scheduler must be able to resolve this condition in a reasonable way (as defined by the system policy on handling overloads).

Given the great significance of application processing cycle management in real-time systems and the primitive state of understanding that exists in this area, it seemed prudent to design the Alpha Scheduling Subsystem to support experimentation with scheduling policies. This would allow many different scheduling algorithms to be tested and evaluated in the process of gaining an improved understanding of their requirements and characteristics. In addition to this, the Archons project's previous work in the area of sched-uling [Jensen 75, Locke 86] has resulted in the generation of a number of requirements for scheduling facilities in operating systems meant to support real-time command and control applications. Included among these are the ability to:

- support a wide range of (possibly unanticipated) scheduling policies, with potentially significant computational demands,
- allow the quick and easy substitution of different scheduling policies into the operating system, transparent to the applications, and
- permit the simple and effective monitoring of the detailed behavior and overall efficacy of a chosen scheduling policy.

In addition to managing application processor cycles, the Scheduling Subsystem should also be involved in the management of other system resources in Alpha. For example, the Storage Management Subsystem might interact with the Scheduling Subsystem in order to decide how best to carry out its function. This is because the Scheduling Subsystem is the repository for all of the system's timeliness information and is optimized for the execution of the policy function. The Scheduling Subsystem contains the time constraint information for each of the computational activities in the system, which includes both the information given by the user and that accumulated by the system at run time. Therefore, the Scheduling Subsystem must be designed to permit other facilities to use its information (concerning timeliness constraints and scheduling policy) to make globally consistent, time-driven resource management decisions.

One function of an operating system is to present an application programmer with a set of abstractions that obscure the unpleasant details of the underlying hardware. In particular, most operating systems permit a programmer to write code with the assumption that it executes alone on a (virtual) hardware base. The more that the realities of a system intrudes on this abstraction (e.g., via interrupts, DMA, etc.), the more difficult the programmer's job and the harder it is for the system to meet the timeliness demands of its application programs. By centralizing the function of managing application processor cycles, Alpha eliminates the conflicts arising from the disjoint management of cycles by the scheduler and the hardware—cycles are to be given out when the scheduler decides it is appropriate, not when they are requested by the hardware.

According to this approach, interrupts that require context swaps to be handled should be routed to the Scheduling Subsystem, allowing it to determine when cycles should be given to the handlers. Taking this concept further, the Scheduling Subsystem should also handle synchronization requests—i.e., when logical synchronization tokens are released (i.e., when V/Unlock operations are performed on semaphore/lock objects) the system should not just select the oldest waiting computation to be allowed to run, but rather it should choose one of the waiting computations based on their current timeliness constraints. This represents one more step towards the unification of all system resource management decisions under a common real-time policy.

The requirements outlined above strongly influenced the design of the Scheduling Subsystem in the Alpha operating system, and a number of features of this subsystem are in direct support of these issues. For example, in order to allow the simple substitution of differing scheduling policies, as well as to facilitate the application of hardware support for scheduling functions, the Scheduling Facility was partitioned from the remainder of the kernel. A clean and well-defined interface has been created between the scheduler and the remainder of Alpha, which has resulted in a number of beneficial side-effects. Included among the benefits of the clear and clean separation of the scheduler from the Kernel Proper are: execution independence—allowing scheduling functions to be performed in parallel with the execution of application activities (in support of the computational demands of real-time scheduling algorithms); a high degree of modularity—permitting the independent modification of the kernel's components (permitting the easy modification of scheduling algorithms); and the codification of the kernel's interface to the scheduler—simplifying the task of monitoring, evaluating, and comparing the behavior of the various algorithms used by the scheduler.

It should be noted that the scheduler is a special operating system facility. Whereas most facilities can be implemented hierarchically (i.e., built out of more primitive, lower-level facilities), the Scheduling Facility is fundamental and presents a cycle that must be broken. The scheduler cannot be implemented as a standard computational unit because it would require that the scheduler be executed in order to run the scheduler—a circularity that must be broken by providing special dispatching services for the scheduling activity.

Because Alpha is a distributed operating system, the local replicas of the scheduler that exist at each node must work together with those at all others nodes in order to ensure the globally consistent management of system-wide processor cycles. This requires that the distributed collection of local/per-node Scheduling Subsystem must be constructed in such a fashion as to share resource management information and a common set of low-level resource management policies.

4.2 Application Time Constraints

The Alpha kernel provide a collection of constructs that permit the programmer to express the timeliness requirements of code regions. These constructs provide the system with the information needed by to perform time-driven management of system resources. The programmer's timeliness needs are expressed to the system in a form known as *time-value functions*, which provide the Scheduling Subsystem with the information necessary to carry out a wide range of scheduling policies. These kernel primitives permit timeliness constraints to be associated with arbitrary regions of code, and in a nested fashion.

The current implementation of Alpha includes time-driven scheduling algorithms that take full advantage of the information provided by time-value functions. Other, more traditional scheduling algorithms make use of a subset of the provided timeliness information.

The distributed nature of the Alpha operating system has a profound impact on the implementation of system timeliness features. This is especially evident with respect to the manner in which time is handled by the system. As is the case with most distributed systems, time must be coordinated among the nodes in Alpha. However, because of Alpha's real-time nature, a special effort must be made to provide nodes with a means of working off a coordinated time-base.

4.2.1 Programming Interface

In Alpha, the programmer indicates to the system his desire that a certain region of code be traversed in a given period of time by enclosing the desired region of code within a pair of begin-/end-time-constraint operations (i.e., via the BEGIN_TIME_CONSTRAINT and END_TIME_-CONSTRAINT standard operations defined on threads).

In order to allow the system to resolve, in a reasonable fashion, situations where the desired time constraints cannot all be met, time constraints in Alpha are defined in terms of time-value functions (which are defined in [Northcutt 88b] and [Shipman 88]). This approach provides the system with application-defined information concerning both the urgency (i.e., the time constraints) and the importance (i.e., the relative value to the system) of regions of code. This permits the system to manage the system resources in such a fashion as to ensure that the maximum number of time constraints can be met given the currently available resources, and when all of the time constraints cannot be met, load is shed in a manner that results in the maximum amount of value being delivered to the system.

Each time a BEGIN_TIME_CONSTRAINT operation is invoked, the kernel evaluates the parameters given with the operation and modifies the thread's state to reflect the newly defined constraint. The time constraints associated with a thread are maintained as a part of the thread's state (known as its *attributes*), and consist of an ordered list of the currently active time constraints (i.e., those constraints which have yet to be met or missed). The programmer may use time constraints in a nested fashion, in which case the system manages the thread's time constraint information in a stack-oriented fashion (pushing and popping time constraints as they are initiated and completed in the course of a thread's execution). Threads executing within nested time constraints operate under the most strict of their currently active time constraints—i.e., an operation to begin a time constraint is effectively ignored should a more demanding time constraint already be in effect. Similarly, when a thread exits a time constraint block (i.e., executes an END_ TIME_CONSTRAINT operation or encounters an exception—e.g., fails to meet a hard deadline or aborts a transaction), the kernel determines whether the thread's environment should be altered to reflect a less constraining outer-level time constraint.

An example time value function and its various components is illustrated in Figure 14. The components of a time value function, and their intended use within the system are defined as follows:

- ***I*** — **Thread Importance:** the relative value of the computation to system. This value is intended to be used in resolving conflicts during overload conditions (i.e., circumstances where there are not sufficient cycles to meet all of the active thread's time constraints).

- t_c — **Critical Time:** the time at which the execution of the defined block of code should be complete. This value is used as the primary determinant of a code block's time constraint.
- t_e — **Expected Execution Time:** the amount of computation time expected to be required by the code block to complete. This value can be used by the system to predict overload conditions.
- $\sigma(t_e)$ — **Standard Deviation of t_e:** an indication of the degree of variance associated with the given estimate of required execution time. This value is to be used as a measure of confidence in the given execution time estimate.
- S_{pre} — **Shape of Value Function, Pre-Critical Time:** an index into a system-generation-time alterable table of pre-computed value curve shapes. This value is used to select the general shape of the time-value function prior to the critical time.
- S_{post} — **Shape of Value Function, Post-Critical Time:** an index into a system-generation-time alterable table of pre-computed value curve shapes. This value is used to select the general shape of the time-value function following the critical time.
- i_{pre} — **Importance Modifier, Pre-Critical Time:** a multiplier used to scale the portion of the chosen time-value curve prior to the critical time. This value can be used to scale the curve in such a manner as to encode the thread's importance into the time-value curve.
- i_{post} — **Importance Modifier, Post-Critical Time:** a multiplier used to scale the portion of the chosen time-value curve following the critical time. This value can be used to scale the curve in such a manner as to encode the thread's importance into the time-value curve.

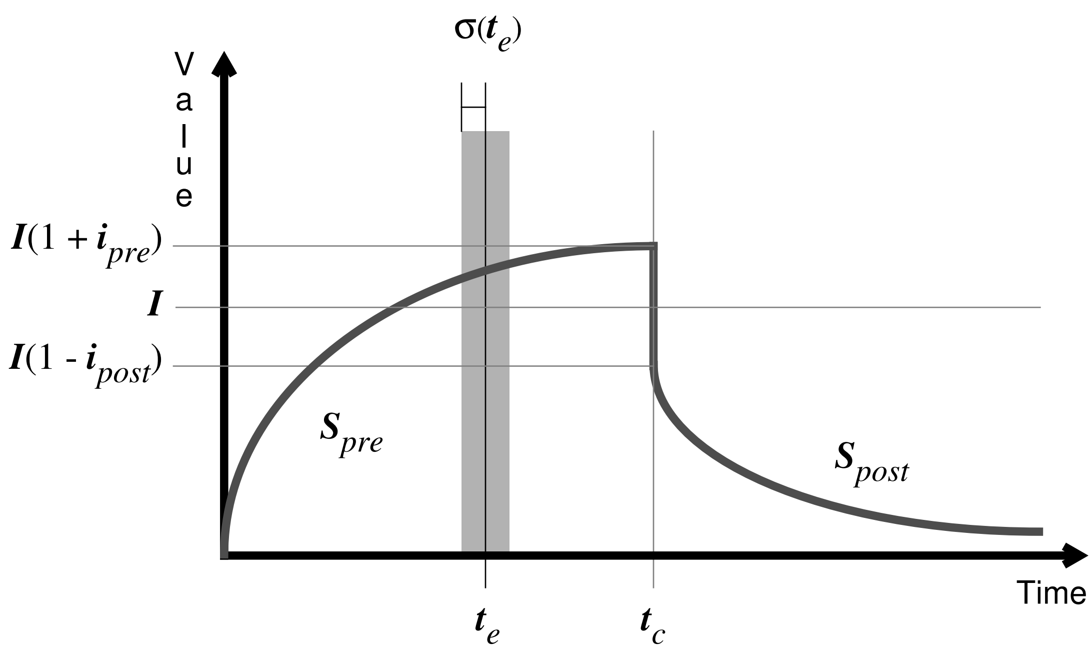


Figure 14: Components of a Time-Value Function

The following time-value function components given as arguments to the BEGIN_TIME_CONSTRAINT operation: $\mathbf{t}_c$, $\mathbf{t}_e$, $\sigma(\mathbf{t}_e)$, $\mathbf{S}_{pre}$, $\mathbf{S}_{post}$, $\mathbf{i}_{pre}$, $\mathbf{i}_{post}$. The arguments $\mathbf{t}_c$, $\mathbf{t}_e$, and $\sigma(\mathbf{t}_e)$ are given in milliseconds, with type **float**. The $\mathbf{S}_{pre}$ and $\mathbf{S}_{post}$ arguments are indices of system-defined value-function shapes, and are of type **unsigned short**. The shapes of value functions currently defined in Release 1 of Alpha are defined in [Shipman 88]. The arguments $\mathbf{i}_{pre}$ and $\mathbf{i}_{post}$ are of type **short** and are expressed as percentages of the thread's importance (**I**), which is determined by invoking an operation on a system service object (as defined in [Northcutt 88d]).

Figure 15 provides an illustration of the use of the kernel's time constraint primitives to implement the Alpha interim object programming language's time constraint block construct. This example illustrates how a time constraint block and its exception handler is implemented—a BEGIN_TIME_CONSTRAINT operation is invoked on the thread when the time block is entered, and at the completion of the execution of the code within the time constraint block an BEGIN_TIME_CONSTRAINT operation is invoked on the thread. When a time constraint block's time-value function crosses the value origin (or falls to within *epsilon* of zero—where *epsilon* is defined as some small value by the chosen scheduling policy), the thread is said to have missed its *deadline*. When a thread invokes the BEGIN_TIME_CONSTRAINT operation, the kernel makes a snapshot of the thread's execution state, and should the thread miss a deadline (or encounter another form of exception) the thread's saved state is restored to the point where the BEGIN_TIME_CONSTRAINT operation was invoked. Normally, the BEGIN_TIME_CONSTRAINT operation returns a boolean value of TRUE when it is invoked, however, when an exception occurs, the operation returns a FALSE. In this way, the exception handling code is entered for each of the (potentially nested) exception blocks, with the all of the thread's state available to it for carrying out the desired clean-up functions.

While the thread is executing the code within the body of a time constraint block it will be scheduled according to the time-value function specified by the arguments provided in the BEGIN_TIME_CONSTRAINT operation. When a thread exits a time constraint block (either as a result of completing the code within the allocated time, or due to missing a deadline), execution continues in the code following the end of the block, it assumes the time constraint attributes that it had prior to entering the block, and is given application processor cycles appropriately.

```
/* define time constraint block */
TIME CONSTRAINT BLOCK (argument_list) {
    /* block of code to which the time constraint applies */
    ...
} ON EXCEPTION {
    /* exception handling code for this time constraint block */
    ...
}
```

```
/* time constraint */
if (SELF.BEGIN_TIME_CONSTRAINT(CriticalTime,
                               ExpectedExecutionTime,
                               ExpectedExecutionTimeVariance,
                               PreCriticalTimeShapeIndex,
                               PostCriticalTimeShapeIndex,
                               PreCriticalTimeImportanceModifier,
                               PostCriticalTimeImportanceModifier) == TRUE) {
    /* block of code to which the time constraint applies */
    ...
    SELF.END_TIME_CONSTRAINT ();
}
else {
    /* exception handling code for this time constraint block */
    ...
    END_EXCEPTION ();
}
```

Figure 15: Time Constraint Block Example

4.2.2 Implementation Issues

When a BEGIN_TIME_CONSTRAINT operation is invoked, the critical time given as a parameter is converted into an absolute time within the local node's time-frame. This indication of time is sufficient for the system's needs as long as the thread to which it applies remains on the node where the time constraint was given. However, when threads move among nodes (either as a result of a remote operation invocation, or the migration of a thread/object), the critical time values must be made to remain valid with respect to its local node's real-time clock. In Alpha, maintaining synchronization among clocks in all of the nodes in the system was considered an unreasonable attempt at simulating the behavior of a centralized system. Thus, it was decided that thread time constraints must be adjusted to each new local time reference as threads move among nodes. In support of this, the time taken by threads as they move among nodes should be provided to the kernel. Despite the fact that the current communication subnetwork in Alpha (i.e., Ethernet) does not support this functionality, the Communication 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 communication 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 Storage Management 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 of execution times. Since real 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.

Nested time constraints are implemented by the queuing of time constraints on a stack of thread timeliness attributes. In the course of such nesting, less demanding time constraints are ignored by the system, however the time attribute stacks are kept consistent by pushing timeliness attributes when BEGIN_TIME_CONSTRAINT operations are invoked, and popping the attributes when END_TIME_CONSTRAINT operations are invoked. Furthermore, the kernel ensures that the consistency of thread timeliness attribute stacks is maintained despite thread exceptions (e.g., missed time constraints and thread failures).

Time constraint blocks are standard Alpha exception blocks—they use the same basic mechanisms that save a thread's state at the beginning of a block, restore it if exception occurs before the block is exited (and go to an exception handler), and discarding the thread state when the block is finally exited.

The state associated with a time constraint block includes: the parameters given in BEGIN_-TIME_CONSTRAINT operation, the run-time statistics accumulated by system, the processor state at point of entry (that can be used for restoring the state of the thread should exceptions occur), and "digested" (i.e., translated/normalized) values passed between nodes on remote operation invocation. Also, the Scheduling Subsystem was implemented in such a manner as to allow the definition of arbitrary time-value curve shapes and assorted probability distributions at system-generation time.

The manner in which both absolute and relative time values are used in Alpha dictates that the need in the Scheduling Subsystem for floating point support. Floating point values are required both for reasons of dynamic range (i.e., very long and very short time constraints) and granularity (i.e., the ability to define times on the desired time boundaries).

4.3 Scheduling Facility Implementation

The Alpha Scheduling Facility is a major component of the kernel—it not only provides a critical system function (i.e., the distribution of processing cycles to computational tasks), but it is also a facility that interacts strongly with the other kernel-level resource management functions (e.g., remote invocation, synchronization, and storage management). The Scheduling Facility's place in Alpha is illustrated in Figure 2, as is the two main components of the facility. The scheduler consists of a component that exists in the Kernel Proper, and a component that executes independently on the Scheduling Co-Processor and is known as the *Scheduling Subsystem*.

The part of the scheduler within the Kernel Proper provides the system with an interface to the Scheduling Subsystem, where the bulk of the scheduling function is performed. The Kernel

---

†Protocol Engine Incorporated's Protocol Engine-based XTP/FDDI communication subnetwork is an example of a network that attempts to meet the needs of real-time systems such as Alpha.

Proper's interface to the scheduler is made through a series of routines that control the state of threads.

The routines that comprise the scheduler interface include:

MakeReady: Adds a given thread into the scheduler's ready queue, and puts the thread into the READY state.

Preempt: Initiates a context swap, putting the currently executing thread into the READY state, and the thread selected from the ready queue by the scheduling algorithm is put into the RUNNING state.

Run: Initiates the execution of the thread selected to be placed into the RUNNING state.

Block: Removes the currently running thread from the scheduler's ready queue, puts the thread into the BLOCKED state. This routine is typically called as a result of P and LOCK operations.

Unblock: Adds a blocked thread to the scheduler's ready queue, forcing the thread from the BLOCKED state back to the READY state. This routine is only called on thread exceptions.

Sleep: Performs a Block on the currently executing thread and executes an Unblock on the thread following a specified amount of time.

Suicide: Executes a Block on the currently executing thread, deallocates the thread, returning it to the INITIAL state.

These routines provide a well-defined, higher-level interface to the Scheduling Subsystem via the IMI interface between the Application Processor and the Scheduling Co-Processor. All of the Kernel Proper's interactions with the Scheduling Subsystem are handled via this interface. Only simple, serial functions are performed in the Kernel Proper portion of the Scheduling Facility, all other functions that may benefit from the information contained within the scheduling policy modules, or from concurrent execution, are executed within the Scheduling Subsystem. The Kernel Proper part of the Scheduling Facility notifies the Scheduling Subsystem of all scheduling activities (e.g., threads blocking or unblocking and changing timeliness constraints). As a result of a thread relinquishing the processor (i.e., blocking), the Kernel Proper begins the execution of the idle thread and awaits a command from the Scheduling Subsystem to dispatch the next thread to be run. In summary, the Scheduling Subsystem provides concurrency and the bulk of the functionality, while all that the Kernel Proper side does is act as an interface, perform thread dispatch functions when instructed to do so by the Scheduling Subsystem, and executes the idle thread when a thread gives up the CPU (i.e., blocks itself).

The scheduler's clients are not explicitly aware of the Scheduling Subsystem, and it may therefore be modified without having to make changes within the rest of the kernel. The Scheduling Subsystem maintains thread run-time execution statistics instead of the Kernel Proper because of the lack of a fully-shared memory interface between the Application Processor and the Scheduling Co-Processor. Because of this architectural configuration and for reasons of performance, the Scheduling Subsystem duplicates some information maintained by the Kernel Proper.

The interface between the Scheduling Facility's two components is detailed in the following section.

4.3.1 Internal Interface

The interface between the scheduler's parts is significant in that it must be designed in such a fashion as to be transparent to the Kernel-Proper-side of the scheduler, while allowing the implementation of any desired policy within the Scheduling Subsystem. This requires that both the commands and the information exchanged via this interface be sufficient and appropriate to support the needs of a wide range of potential scheduling policies. It is believed that the interface in the current implementation of Alpha meets these needs, and the following provides a definition of the logical interface and details of the actual interface.

The interface to the Scheduling Subsystem was designed so that the Scheduling Facility can perform its function external to the application processor that it is managing. The interface is composed of a collection of commands that are exchanged via a node's interprocessor communication mechanism (i.e., the kernel's IMI service).

In the course of system operation, the Scheduling Subsystem may issue a command to the Kernel Proper, indicating that the currently executing thread should be preempted and a new thread is to be dispatched. The preemption command includes a reference to the Alphabit identifier of the new thread to be executed. The Kernel Proper responds to such a preemption command by suspending the execution of the currently active thread, performing a context swap to allow the execution of the indicated thread, and issuing an acknowledgment to the Scheduling Subsystem indicating that the thread has been suspended. When the Scheduling Subsystem receives a preemption acknowledgment, it adjusts the accumulated computation time for the newly suspended thread, and continues with its scheduling functions while the Application Processor executes the newly dispatched thread.

The scheduler's internal interface also includes commands that allow the Scheduling Subsystem to be notified of changes in a thread's state. These commands are used to indicate when a thread should be added to, or removed from, the scheduler's list of currently active threads (i.e., the *ready queue)* and include information concerning the reason for the indicated manipulations—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 Kernel Proper to inform the Scheduling Subsystem of changes that occur in the attribute information of one of the threads currently in the *ready queue*.

The Scheduling Subsystem's interface has also been designed to permit the Scheduling Facility to execute entirely within the Kernel Proper, should the chosen scheduling algorithm not be able to take advantage of the concurrency provided by the Scheduling Subsystem (such as might be the case with simple round-robin or priority scheduling schemes). Additionally, the Scheduling Subsystem can use the thread run-time information that it maintains 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.

The actual interface between the Scheduling Facility's components is illustrated in Figure 16.

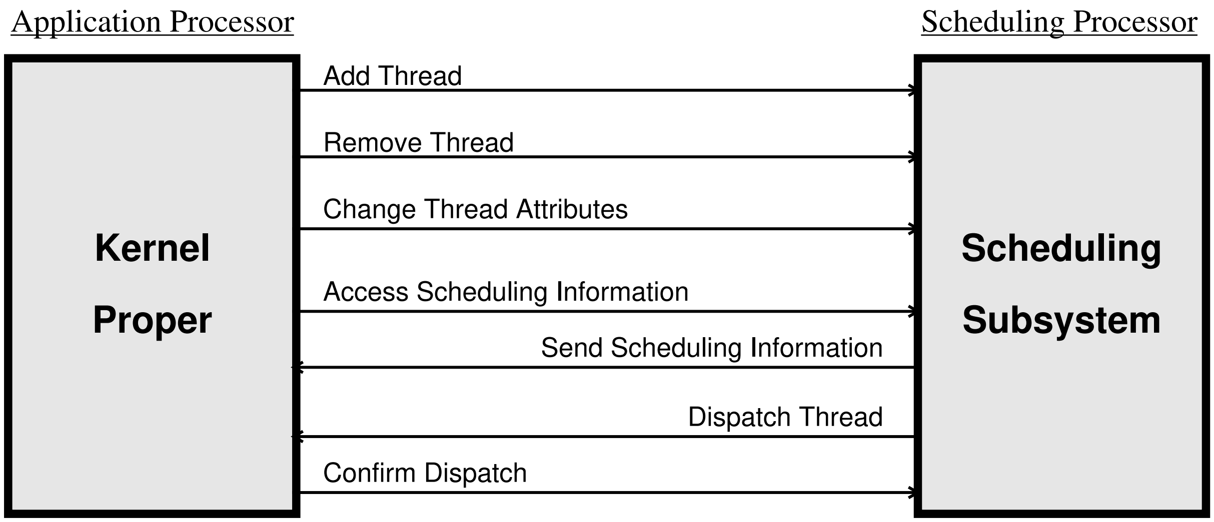


Figure 16: Application Processor/Scheduling Subsystem Interface

The commands sent from the Kernel Proper to the Scheduling Subsystem are:

- **Add Thread**: This command takes as parameters the Alphabit identifier for the thread to be added to the scheduler's ready queue, along with the necessary scheduling information, and sends a message containing these parameters to the Scheduling Co-Processor.

  There are three variations of this command—the first being the addition of a simple thread to the ready queue (i.e., due to a local thread unblocking), the second being the addition of a surrogate thread (i.e., a thread created locally to perform a remote invocation from another node), and the third being an add command that does not take effect until after a specified amount of time has elapsed (i.e., a delayed add).

  The first time that a thread is added to the ready queue, its scheduling information is passed to the Scheduling Subsystem with the add command. The Scheduling Subsystem retains this information until the thread is destroyed, therefore subsequent add commands need only include the thread's Alphabit identifier.

  The add operations performed on surrogate threads take as parameters scheduling information in the Scheduling Subsystem's internal format. This is because the thread has already been executed (albeit on another node) and has accumulated statistics. Also, the thread's scheduling information has already been given to the Scheduling Subsystem and converted from the format it was provided from the user in, to the format that the subsystem maintains.

- **Remove Thread**: This command takes as a parameter the Alphabit identifier of a thread to be removed from the ready queue, and sends it in a message to the Scheduling Subsystem.

  There are two varieties of this command—one that simply removes a given thread from the scheduler's ready queue, and another that removes the thread from the ready queue and deletes the scheduling information associated with it from the Scheduling Subsystem's storage.

- **Change Thread Attributes**: This command alters the time constraint information associated with a thread by the Scheduling Subsystem. There are three types of change commands—one to push new time constraints on a thread's time constraint stack, a second command to pop them off, and yet another command to change the global scheduling attributes of a thread.

  The command to push time constraints takes as parameters a thread's identifier and the time constraint information as given by the client, and passes them to the Scheduling Subsystem in messages. While the command to pop time constraints takes as a parameter a count indicating the depth to which the thread's time constraint stack is to be popped to. Both commands take a parameter that indicates the thread which is to be changed.

  The command to change a thread's global scheduling attributes takes an identifier for a thread and the new scheduling attributes to be applied to the thread (e.g., the thread's importance), and passes them in a message to the Scheduling Subsystem where they are applied to the given thread's control block.

- **Access Scheduling Information**: There are two different varieties of this command—one to load, and another to store, the Scheduling Subsystem's information related to a thread's current state (i.e., run-time statistics, normalized/transformed scheduling parameters, top of time constraint stack, etc.).

  Both types of this command pass as parameters a thread's Alphabit identifier, and a pointer to a shared memory location (outside of the IMI channel's space) where the scheduling information is to be loaded from or stored to.

- **Confirm Dispatch**: This command is a simple message to the Scheduling Subsystem indicating that the requested thread dispatch has been performed. It requires no parameters, and is used to synchronize the Kernel Proper and Scheduling Subsystem. Such synchronization is necessary because the currently running thread may choose to release the processor at the same time that the Scheduling Subsystem issues a preempt command.

  This command also allows the Scheduling Subsystem to maintain accurate run-time statistics for the threads. Which is necessary because the time required to perform a preemption may vary—e.g., preemption may be briefly deferred for kernel synchronization purposes, or context swaps may involve differing amounts of effort depending on the type of threads being swapped (i.e., kernel or client).

The commands issued by the Scheduling Subsystem to the Kernel Proper are:

- **Dispatch Thread**: This command sends the Alphabit identifier of the thread that the Scheduling Subsystem's policy determines should be bound to an Application Processor. This command instructs the Kernel Proper to suspend the currently executing thread and run the thread given as a parameter to the command .

  A confirmation message is expected from the kernel for this command.

- **Send Scheduling Information**: This command is used to signal that the requested scheduling information has been stored into the buffer indicated in a proceeding command to access the scheduling information for a thread.

  In effect, this command serves as a synchronization message that permits the Kernel Proper to use the scheduling information that it has requested (e.g., a thread's scheduling information for shipment to another thread on a remote invocation.

As an optimization, the components of the Scheduling Facility exchange local control block pointers, in addition to the globally unique Alphabit identifiers used to indicate the thread which is the target of a particular operation. This enhances the scheduler's performance by simplifying the identification of thread control information on both sides of the interface.

4.3.2 Internal Structure

A primary design objective of the Scheduling Facility in Alpha was the ability to support a wide range of (possibly experimental) real-time scheduling policies. This objective was achieved in Release 1 of Alpha by way of a logical framework for the Scheduling Subsystem within which a wide range of differing scheduling policies can be implemented, through the substitution of policy-specific routines within a well-defined structure. The Scheduling Subsystem framework consists of a collection of scheduling mechanisms and the interface definition for a collection of routines that implement various aspects of scheduling policies. In the current implementation of the Scheduling Subsystem, multiple policies can be implemented by inserting a suite of policy routines into the scheduling mechanism framework, and then selecting the desired policy (by way of link- or run-time setable switches).

The overall design of the Scheduling Subsystem is one in which all activity is performed by policy routines, in response to the receipt of *tokens*. Each scheduling policy is composed of a series of actions that are performed in response to certain (internally or externally generated) stimuli. In this design, all hardware interrupts (which stem from either the scheduling co-processor's on-board devices or messages from the Application Processor) and inter-policy-routine communications are manifest as tokens, each of which is directed to a specific target policy routine. The receipt of a token by a policy routine triggers its execution, the result of which may be the modification of the scheduler's state or the generation of additional tokens.

Figure 17 provides an illustration of the logical structure of the Scheduling Subsystem in Release 1 of Alpha.

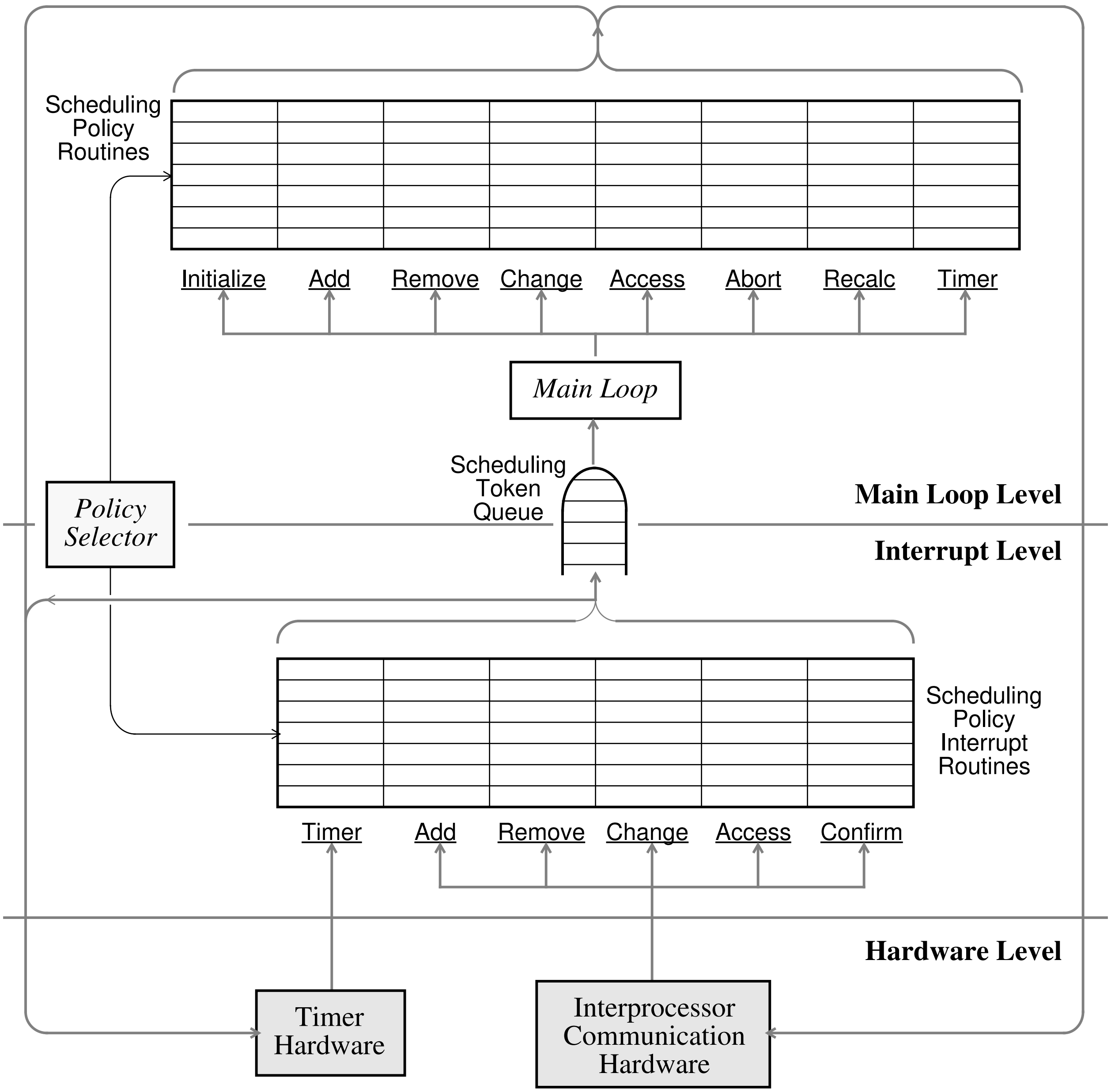


Figure 17: Scheduling Subsystem Internal Structure

The Scheduling Subsystem is composed of two levels—the *interrupt level* and the *main loop level*—each of which is composed of a collection of routines that carry out specific aspects of a total scheduling policy. As its name implies, the interrupt level contains routines that handle the tokens generated (by the underlying framework) as a result of hardware interrupts. The main loop level contains routines that handle tokens that were generated by other routines (at either the main loop or interrupt levels). The substrate upon which the Scheduling Subsystem is constructed provides the basic services of converting interrupts into tokens, managing the queueing and distribution of tokens, and controlling the execution of the routines specified by the targets of tokens.

The Scheduling Subsystem's framework provides: the overall structure to scheduling policies (by defining the token handling routines and their general function); the token handler indirection tables (used to map scheduling tokens to the proper routines for the chosen policy); the interrupt-token generation code (for the IMI interface from the Application Processor and the on-board interval timer); scheduling token manipulation routines (e.g., allocation/deallocation, and scheduling token queue insertion/removal mechanisms); the main loop token interpreter (that removes tokens from the queue and dispatches them to the appropriate token handler routine); a collection of general purpose monitoring and debug mechanisms, hardware management routines (e.g., memory management, timer management, and low-level I/O); the IMI interprocessor communication package (for interacting with the Applications Processor); and a set of stand-alone math libraries (providing support for floating point and probability distribution manipulations).

In order to simplify the synchronization concerns of policy writers, an effort was made in the implementation of the Scheduling Subsystem's mechanisms to ensure that the proper (i.e., FIFO) sequencing of tokens within the main loop queue is maintained in the face of multiple, asynchronous insertions and removals of tokens. Also, default handler routines are placed into indirection table slots where no such scheduling token is expected for the given policy. These default routines signal an exception condition and can optionally be made to generate a system fatal error. In addition to these routines, there can also be a set of custom debug routines defined for each policy. These debug routines can be used by systems developers to print out the (formatted) contents of (both general and policy-specific) data structure, dynamically collected run-time statistics, and other scheduling-policy-specific information.

The following text describes the mechanisms provided by the Scheduling Subsystem's framework and each of the its token handling routines, starting with the interrupt level ones and followed by the main loop level routines.

4.3.2.1 Scheduling Subsystem Mechanisms

A collection of general-purpose mechanisms are provided as part of the Scheduling Subsystem's framework. These mechanisms are provided for use by the routines that make up individual scheduling policies, and are grouped into functionally related sets known as *facilities*. The Scheduling Subsystem's facilities provide mechanisms for the management of memory, time, tokens, and interprocessor communication. The support mechanisms provided by each of these Scheduling Subsystem facilities are defined as follows†:

Memory Management Facility

- Dynamic Memory Allocator

INTERFACE: vaddr Falloc()

PARAMETERS: none

RETURNS: Pointer to location in memory of newly allocated block, or a 'NULL' pointer if there is insufficient free memory to honor the request.

DESCRIPTION: This routine returns a pointer to a fixed size block of memory. The block size is equal to that of a data structure that is the union

†*The various initialization routines associated with the facilities are omitted here for reasons of brevity.*

of all of the data structures used by scheduling policies. The allocator was designed, above all else, to be fast.

- Dynamic Memory Deallocator

INTERFACE: void Fdealloc(vaddr fptr)

PARAMETERS: The fptr parameter points to the block of memory to be deallocated.

RETURNS: nothing

DESCRIPTION: This routine takes a pointer to a previously allocated block of memory and adds it to the allocator's free list. For performance reasons, no check is made to see if the pointer refers to a valid, currently allocated block of memory.

Time Management Facility

- Create a Timed Event

INTERFACE: short CreateTimedEvent(u_long interval, tokenptr tknptr)

PARAMETERS: The interval parameter indicating an amount of time in terms of scheduling processor clock ticks. The tknptr parameter is a pointer to a scheduling event token.

RETURNS: An identifier for the timed event, or a 'NIL' value if it is unable to allocate a timed event.

DESCRIPTION: This routine takes an indication of the amount of time until a timed event is to occur, and a pointer to the token that is to be issued when the given interval of time has elapsed.

- Cancel a Timed Event

INTERFACE: void CancelTimedEvent(u_short event)

PARAMETERS: The event parameter is an identifier for a timed event.

RETURNS: nothing

DESCRIPTION: This routine deletes a timed event indicated by the identifier for the timed event that was returned when it was created. This routine has no effect if the indicated timed event has already expired when it is invoked.

- Read the System Clock

INTERFACE: u_int GetSystemTime()

PARAMETERS: none

RETURNS: Integer representing current time.

DESCRIPTION: This routine returns a value that indicates the current time in terms of system clock ticks. One tick is equal to 3.2 microseconds.

Token Management Facility

- Create a Scheduling Event Token

INTERFACE: tokenptr CreateToken(tokentype type, tokenargs args)

PARAMETERS: The type parameter is an enumerated type defining the types of tokens that exist. The args parameter is a union of the data structures that comprise the data part of the possible scheduling tokens.

RETURNS: Pointer to the newly created and initialized scheduling event token.

DESCRIPTION: This routine takes an indication of the desired token's type and the data associated with the token, allocates the memory for a token, initializes the token's body with the given data, and then loads the current system time into the token's header.

- Delete a Scheduling Event Token

INTERFACE: void DeleteToken(tokenptr tknptr)

PARAMETERS: The tknptr parameter is a pointer to a scheduling token.

RETURNS: nothing

DESCRIPTION: This routine takes a pointer to a token and deallocates it, returning its memory to the free pool.

- Enqueue Scheduling Event Token

INTERFACE: void EnqueueToken(tokenptr tknptr)

PARAMETERS: The tknptr parameter is a pointer to a scheduling token.

RETURNS: nothing

DESCRIPTION: This routine inserts a token into the tail of the scheduling event token queue, to processed by the main loop.

Interprocessor Communication Facility

- Send a Message to the Application Processor

INTERFACE: void SendMsg(short opcode, message msg)

PARAMETERS: The opcode parameter is an constant indicating the type of parameter to be transmitted. The msg parameter is a union of the types of messages that can be sent, and it contains the text of the message.

RETURNS: nothing

DESCRIPTION: This routine takes an indication of the type of message to be sent and the body of the message as parameters. It composes a message and transmits it to the Application Processor via the Interprocessor Message Interface.

4.3.2.2 Interrupt Level Routines

Whenever an interrupt occurs at the SP, a token is generated and passed to an interrupt handling routine (as indicated by the policy indirection tables). The scheduler's substrate handles both the generation of these tokens, therefore there is room for alternative interpretations of the function of these routines (e.g., the **Timer** routine is only invoked as an interrupt generated by the on-board timer).

The functional descriptions for the Scheduling Subsystem's interrupt level scheduling policy routines are defined as follows:

- **Timer**: This routine receives tokens from the on-board counter/timer unit, which is configured in the Scheduling Subsystem to function as an interval timer. The tokens that this routine receives were generated by other routines at an earlier point in time, and inserted into the timed event queue maintained by the interval timer facility.
- **Add**: This routine receives tokens from the IMI facility, which are generated as a result of receiving Add_Thread commands from the Kernel Proper. If a delayed add is desired, this routine can generate a timed event (reusing the token that it receives). Otherwise, the token can be inserted into the scheduling event queue for processing at the main loop level.
- **Remove**: This routine receives tokens from the IMI facility, which are generated as a result of receiving Remove_Thread commands from the Kernel Proper. Generally, the received token is inserted into the scheduling event queue for processing at the main loop level.
- **Change**: This routine receives tokens from the IMI facility that indicate the desire to change the scheduling information associated with a thread. The typical function of this routine is to enqueue the received token into the scheduling event queue.
- **Access**: This routine receives tokens from the IMI facility that indicate the desire to load or store the currently relevant scheduling parameters associated with a thread. The typical function of this routine is to enqueue the received token into the scheduling event queue.
- **Confirm**: This routine handles the preemption confirmation commands that are returned by the Application Processor in response to commands issued to it earlier. This routine provides the functions required to maintain the synchronization between the components of the Scheduling Facility in the Kernel Proper and the Scheduling Subsystem. Typically, all that this routine does is update the Scheduling Subsystem's state to indicate the identity of the currently executing thread, and update the run time statistics being kept for the thread that was preempted. In order to help detect when the running thread has released the Applications Processor at the same time that a Dispatch command is sent, a distinction is made by the Kernel Proper between preemptions of the idle thread, and all other threads. With this information, synchronization can be maintained among the scheduler parts.

In addition to the example functions defined for each of these routines, the desired statistic gathering activities may also be performed in these routines. Information gathered in this way is typically used by the debug routines added as part of a scheduling policy. It should also be noted that any of these routines can be a *null* routine—i.e., a fatal error is generated should this type of interrupt occur. Furthermore, the scheduling token that initiated the execution of an interrupt routine may either be deallocated (when the interrupt routine handles the event entirely at interrupt level), or reassigned and inserted into the scheduling event queue (to be handled at the main loop level). An scheduling token may alternatively be placed into the interval timer queue, to appear later in the form of a token passed to the specified routine. This action has the effect of delaying the arrival of a token for a specified period of time.

4.3.2.3 Main Loop Level Routines

Tokens are queued up in the scheduling event queue by the interrupt handling routines and the main-loop token policy routines. The function of the main loop is to repeatedly remove tokens from the queue, dispatch them to the appropriate policy routines, and initiate the execution of the target routines.

The functional descriptions for the Scheduling Subsystem's main loop level scheduling policy routines are defined as follows:

- **Initialize**: This routine is intended to provide the initialization functions for the specific policy being used. On system start-up, the Scheduling Subsystem's framework generates a token targeted to this routine. This routine is typically executed once at system start-up time, and is not executed again. The functions that may be performed by this routine include the allocation and initialization of data structures, the creation of initial timed events (e.g., for initiating time-slicing activities), and signalling the Scheduling Subsystem's readiness to the Kernel Proper. It should be noted that the Scheduling Subsystem's mechanisms are initialized at system start-up time, therefore the policy module's routines need only initialize the policy-specific aspects of the subsystem.
- **Add**: The intent of this routine is to add a given thread to the set of active threads maintained by the Scheduling Subsystem. For convenience, this collection of threads is known as the *ready mix*, regardless of the actual manner in which the scheduling policy manages the set of threads. In addition to simply adding a new thread to the ready mix, this routine is frequently involved in the process of ordering the threads in a *ready queue*. When a thread is added to the ready mix, the ready queue may be reordered followed by a check to determine if a currently running thread should be halted and a different thread dispatched. Alternatively, this routine may just add a thread to the ready mix, and then generate a token to perform the ready queue reordering and thread dispatch check.
- **Remove**: This routine is used to remove a thread from the ready mix. This is done when a running thread gives up the processor (e.g., is blocked or destroyed), and may require that the ready queue be reordered and reevaluated to determine the next thread to be dispatched. When the running thread blocks, the Scheduling Subsystem may choose to retain the control blocks allocated for the thread and the scheduling information associated with it. This would be done in order to expedite the process of unblocking the thread. In the event that the running thread commits suicide, the token received by this routine indicates this, and the routine may choose to deallocate the state associated with the thread within the Scheduling Subsystem.
- **Change**: This routine is to be used to modify the state information kept by the Scheduling Subsystem for the threads it is aware of. The types of modifications that can be performed are the changing of the thread's state (e.g., its importance value), the pushing of new time constraints, and the popping of old time constraints. The changes made to a thread's scheduling state by this operation may require the reordering of the ready queue and the dispatching of new threads.
- **Access**: This routine is intended to be used to load and store the collected internal state of threads. This function is needed to support the movement of thread scheduling state information among nodes in the distributed system. When a thread leaves a

node on a remote invocation, the thread's scheduling state is obtained and passed with the thread. At the remote node, a surrogate thread is created and initialized by installing the passed scheduling state into the remote node's Scheduling Subsystem.

- **Abort**: The intended use of this routine is in dealing with an exception condition that relates to the scheduling state of a thread in the ready mix. This routine is typically executed as a result of a token generated by a timed event that signals the fact that a thread's critical time has passed. For example, this routine may handle the case where a thread misses a hard deadline by popping the thread's time constraints, reestablishing the thread's previous time constraints, and initiating a reordering and reevaluation of the ready queue. In addition, this routine may signal the Kernel Proper that a thread has failed to meet a deadline, which may in turn cause the Communications Subsystem to initiate a thread abort sequence among the nodes involved with the thread.
- **Recalc**: This routine is intended to be used to reorder and reevaluate the ready mix in order to determine whether a new thread should be dispatched. This routine is typically executed following a scheduling event that modifies the state of the ready queue, or when a timed event indicates a potentially significant point in time has arrived. This routine is intended to be used when the reordering function cannot be performed in a reasonable amount of time by another routine. With this routine, the primary activities of the other routines can be carried out without the additional performance penalty of a (potentially significant) ready queue reordering activity.
- **Timer**: This routine can be used to perform functions associated with timed events in the main loop. This routine is typically not used, as most timer-related activities can be performed at the interrupt level.

4.4 Scheduling Policies

Scheduling policy modules are composed of a collection of routines and data structures that implement the desired scheduling policy within the framework of the Scheduling Subsystem's mechanisms. While the previous subsections described the intended function of each of the different types of routines that make up a policy module, there is no a priori restriction on what functions the routines can perform, nor are there any constraints on their size or organization. The scheduling routines can make use of the mechanisms provided by the underlying subsystem, and can be arbitrarily complex in both structure and behavior.

The following text provides a general description of the scheduling policy routines, and an example of an actual policy module.

4.4.1 Policy Definition Modules

A scheduling policy is defined by a pair of arrays of routines (that are inserted into the interrupt- and main-loop-level policy indirection tables), and a set of data structures (that are used within and among the policy routines).

A complete scheduling policy consists of a set of interrupt handling routines that respond to each type of interrupt that might occur in the course of this policy's execution (e.g., a message arrives from the Application Processor, or the interval timer indicates that a given period of time has elapsed). An interrupt handling routine can (at interrupt level) perform the functions required by the policy in response to a particular interrupt, it can simply generate a token and queue it for processing at the main loop, or an interrupt handling routine can perform some part of the desired function at interrupt level and pass the rest off to be handled by the main loop.

The general intent is that interrupt handlers perform a minimum amount of necessary function, but rather just queue up a token and allow the main loop's token handling routines to perform the bulk of the work necessary for a given interrupt.

At any one time, the Scheduling Subsystem may contain multiple, complete scheduling policies. This is due to the fact that the routine indirection tables that provide the structure for the creation of scheduling policies, are multidimensional—i.e., there are multiple complete sets of token handling routine slots in each table. A particular policy is selected from the current set by way of a *policy selector* variable which serves as an index into the arrays of indirection tables. The system's entire scheduling policy can be changed at run-time by altering the value of the policy selector.

4.4.2 Policy Example

The following is pseudo-code description of the data structures and routines that make up the policy module for an enhanced dynamic deadline scheduling policy.

4.4.2.1 Data Structures

Thread Control Blocks: per-thread data structures that act as repositories for the information that the scheduler needs to know about a thread. They are created when a thread is first introduced to the scheduler, and are deleted when the thread is killed.

Figure 18 illustrates the structure and contents of a thread control block in Alpha.

AP Thread Control Block Pointer
Thread Alphabit Identifier
Thread Importance
Link to Time Constraint Stack
Time Constraint Depth Count
Blocked Flag
Total Accumulated Computation Time
Last Dispatch Time
Importance Queue Header
Deadline Queue Header

Figure 18: Thread Control Block

Time Constraint Blocks: scheduling parameter control blocks that contain information concerning a thread's time constraints. A time constraint block is created each time a thread enters a time constraint block, and is destroyed when the thread exits the block (either normally or as a result of an exception). Each thread control block contains a list of the time constraint blocks for the thread, which is managed in a stack-oriented fashion.

Time Constraint Stack Header
Time at which Command was Issued
This Constraint's Critical Time
Mean Expected Computation Time
Compute Time Standard Deviation
Previously Available Slack Time
Computation Time Left
Computation Time Received
Beginning Exception Block Level
Timed Event Descriptor

Figure 19: Time Constraint Block

Scheduling Queues: linked list of the control blocks for currently active threads. There are two types of queues in this policy module. The first type of thread queue is known as the *deadline* queue, it contains control blocks for ready threads that currently have time constraints and is kept sorted in shortest-time-constraint-first order. The second type of thread queue is known as the *importance* queue and contains the control blocks of the ready threads that currently do not have time constraints associated with them. Furthermore, the thread control blocks in the importance queue are kept sorted in a greatest-importance-first order.

Scheduling Statistics: a collection of counters and accumulators that keep track of the number of times that an event of interest (e.g., a call to a policy routine) occurs and the accumulated sizes of the thread control block queues. This information is used by the debugging code to print the number of times each type of scheduling event occurs, and to print the average and maximum lengths of the scheduling queues.

4.4.2.2 Interrupt Level Routines

Timer: handles interrupts generated by the interval timer facility. There are two reasons why interval timers are used—delayed Add commands and missed hard deadlines. This is to say that either Add or Abort tokens are queued up for the main loop by this routine.

```
DLTimerIntr() {
        /* add a time-stamp to the token */
        TokenPtr->CurrentTime = GetSystemTIme();

        /* queue it up in the event queue */
        EnqueueToken(TokenPtr);

        /* do not dealloc the token yet */
        return(NO_DEALLOC);

}
```

Add: handles interrupts from the Kernel Proper part of the Scheduling Facility that indicate the desire to (either immediately or following a delay) add a thread to the ready mix. If it is a delayed command, then this routine passes the incoming token to the interval timer to be delayed and then issued at a later time. If it is an immediate command, the token is queued up to be handled by the main loop.

```
DLAddIntr() {
        /* check if it is a delayed command */
        if (token is delayed) {
                /* it is, so send it to the interval timer */
                CreateTimedEvent(TokenPtr->DelayInterval, TokenPtr);
        }
        else {
                /* it is not, so queue it up in the event queue */
                EnqueueToken(TokenPtr);
        }

        /* do not dealloc the token yet */
        return(NO_DEALLOC);
}
```

Remove: handles interrupts from the Kernel Proper part of the Scheduling Facility that indicate the desire to remove a thread from the ready mix. This routine just queues up the token to be handled at the main loop level.

```
DLRemoveIntr() {
        /* queue the token up in the event queue */
        EnqueueToken(TokenPtr);

        /* do not dealloc the token yet */
        return(NO_DEALLOC);
}
```

Change: handles interrupts from the Kernel Proper part of the Scheduling Facility that indicate the desire to change a thread's scheduling parameters. This routine just queues up the token to be handled at the main loop level.

```
DLChangeIntr() {
        /* queue the token up in the event queue */
        EnqueueToken(TokenPtr);

        /* do not dealloc the token yet */
        return(NO_DEALLOC);
}
```

Access: handles interrupts from the Kernel Proper part of the Scheduling Facility that indicate a desire to remove a thread from the ready mix. This routine just queues up the token to be handled at the main loop level.

```
DLAccessIntr() {
        /* queue the token up in the event queue */
        EnqueueToken(TokenPtr);

        /* do not dealloc the token yet */
        return(NO_DEALLOC);
}
```

Confirm: handles interrupts from the Kernel Proper which indicate that the requested preemption of the currently running thread has completed. The time is noted when a thread is dispatched, so that when a preempt confirmation command arrives, the amount of computation time received by the thread can be determined. This allows the run-time statistics for the preempted thread to be accurately maintained.

```
DLConfirmIntr() {
        /* check if the preempted thread is not the idle thread */
        if (not preempting idle thread) {
                /* not idling, so update the preempted thread's stats */
                ...
        }

        /* dealloc the token */
        return(DEALLOC);
}
```

4.4.2.3 Main Loop Level Routines

Initialize: initializes the policy module. This includes the creation and initialization of two queues that contain the threads in the ready mix (i.e., the deadline and importance queues), it sets the current thread pointer and preempting thread pointer to NULL, it indicates that a preemption is currently not in progress, and it initializes the timed event facility.

```
DLInit() {
        /* initialize ready queues */
        QInit(IM_Queue);
        QInit(DL_Queue);

        /* initialize global variables */
        currentthreadptr = NULL;
        preemptthreadptr = NULL;
        preemptinprogress = FALSE;

        /* initialize the timed event facility */
        InitTimedEvents();
}
```

Add: adds a thread to the ready mix. This command's token can come directly from a "MakeReady" command in the Kernel Proper, or from the timed event facility as a result of a delayed Add command. This routine can add to the ready mix threads rooted at the local node, or threads rooted at another node (i.e., surrogate threads). If the thread added is locally rooted, it may have already be introduced to the Scheduling Subsystem, or it may be new to the scheduler (i.e., either a newly created thread rooted at the local node, or a surrogate thread). When a thread is first introduced to the Scheduling Subsystem, a thread control block is created and initialized with the thread's initial parameters or those passed with it by the remote node.

If a surrogate thread is under a time constraint, a time constraint control block must also be allocated and initialized with the passed information, and a timed event is created for the thread's currently dominant deadline. If the thread to be added is already known to the scheduler, a pointer to its thread control block is given to the Scheduling Subsystem and the control block is added to the appropriate ready list. Threads are added to the importance or deadline queues (depending on whether they have active time constraints or not) in such a way as to maintain the desired ordering. Finally, a new thread is dispatched if necessary, and the incoming token is deallocated.

```
DLAdd() {
        /* check if this is a new thread */
        if (new thread) {
                /* it is, so allocate and initialize a TCB */
                TCBptr = (tcb)Falloc();
                …

                /* see if it is a surrogate (with passed scheduling info.) */
                if (surrogate thread) {
                        /* it is, so inherit all of its scheduling info */
                        …
                }
        }

        /* prepare the TCB to run */
        TCBptr->blocked = FALSE;

        /* decide if it belongs in the importance queue */
        if (no time constraint) {
                /* no TC, so add it to the importance queue */
                QInsert(IM_Queue, TokenPtr);
        }
        else {
                /* it has a TC, so add it to the deadline queue */
                QInsert(DL_Queue, TokenPtr);
        }

        /* check if we need to dispatch a new thread */
        if (new thread is at head of queue) {
                /* the new thread is at the head of the queue */
                DispatchThread();
        }

        /* deallocate the token */
        DeleteToken(TokenPtr);
}
```

Remove: removes a thread from the ready mix. This command can only come from the currently running thread on the Application Processor. There are two cases to be considered—one where the current thread has decided to block, in which case it is just removed from the ready mix and the next thread is dispatched. The other case is where the thread blocks at (approximately) the same time that the Scheduling Subsystem issues a preempt command, in which case the Scheduling Subsystem's state must be kept consistent with that of the Kernel Proper.

In any event, the given thread is removed from the ready mix, the blocked thread's run-time statistics are updated, and the next ready thread is dispatched. Also, there are two variants of this command—one where the indicated thread control block is removed from the ready queue it is in (i.e., blocking the thread by the Kernel Proper's "Block" command), and another where the control block is removed from the ready mix, then deallocated (i.e., the thread is killed by the "Suicide" command in the Kernel Proper). The control blocks for blocked threads are allowed to float in the Scheduling Co-Processor's memory, while references to them are kept in the Application Processor.

```
DLRemove() {
        /* update the current thread's accumulated computation time */
        …

        /* remove the thread from the ready mix */
        if (under time constraint) {
                /* remove thread from deadline queue */
                QRemove(DL_Queue, TokenPtr);
        }
        else {
                /* remove thread from importance queue */
                QRemove(IM_Queue, TokenPtr);
        }

        /* mark thread as blocked */
        TCBptr->BlockedFlag = TRUE;

        /* check if currently running thread is the one being blocked */
        if (TCBptr is CurrentThreadPtr) {
                /* it is, so dispatch a new thread */
                DispatchThread();
        }

        /* check if the thread control block is to be deallocated */
        if (kill thread) {
                /* it is, so get rid of time constraint stack */
                while (time constraint blocks left) {
                        /* deallocate a time constraint block */
                        Fdealloc(TCptr);
                }

                /* deallocate the thread control block */
                Fdealloc(TCBptr);
        }

        /* deallocate the token */
        DeleteToken(TokenPtr);
}
```

Change: handles the pushing and popping of a thread's time constraints as well as the changing of the thread's importance. When the thread's scheduling information is changed, its control block is removed from whatever ready queue it might be in, changing the state, and then reinserting it into the appropriate queue (in the proper order). If after the thread's state has been changed and it has been reinserted into a ready queue, a dispatch command may have to be issued to run another thread. When time constraints are pushed, new time constraint control blocks are allocated, initialized with the given time constraint information, and pushed onto the head of the time constraint stack. When a pop time constraint is done, the block at the top of the time constraint stack is removed and deallocated.

```
DLChange() {
        /* remove the thread from the ready mix */
        if (under time constraint) {
                /* remove thread from deadline queue */
                QRemove(DL_Queue, TokenPtr);
        }
        else {
                /* remove thread from importance queue */
                QRemove(IM_Queue, TokenPtr);
        }

        /* decide what kind of change is to be made */
        switch (TokenPtr->SubType) {
                case CHANGE: {
                        /* change thread importance */
                        …
                }
                case PUSH_TC: {
                        /* push a time constraint */
                        TCptr = (tc)Falloc();
                        …
                }
                case POP_TC: {
                        /* pop a time constraint */
                        …
                        Fdealloc(TCptr);
                }
                …
        }

        /* reinsert the thread into the appropriate queue and location */
        if (no time constraint) {
                /* no TC, so add it to the importance queue */
                QInsert(IM_Queue, TokenPtr);
        }
        else {
                /* it has a TC, so add it to the deadline queue */
                QInsert(DL_Queue, TokenPtr);
        }

        /* check if we need to dispatch a new thread */
        if (different thread at head of queue) {
                /* the new thread is at the head of the queue */
                DispatchThread();
        }

        /* deallocate the token */
        DeleteToken(TokenPtr);
}
```

Access: allows access to a thread's scheduling information from the Kernel Proper. There are two types of accesses that can be performed as part of this command—i.e., scheduling information reads and writes. For read scheduling information commands, the Scheduling Subsystem places the information for the given thread into the buffer location in common memory indicated in the command from the Kernel Proper. Similarly, writing scheduling information into a thread's control block is done from the shared memory buffer referred to in the command.

```
DLAccessSI() {
        /* check if it is a read command */
        if (dump subcommand) {
                /* it is a read, so copy the information into the buffer */
                …

                /* send a message to the AP to indicate the dump is done */
                SendMsg(SI_DUMP_DONE, NULL);
        }
        else {
                /* it is a write, so copy the information out of the buffer */
                …

                /* see if the thread's importance has changed */
                if (importance changed) {
                        /* adjust the thread's location in the IM queue */
                        QRemove(IM_Queue, TokenPtr);
                        QInsert(IM_Queue, TokenPtr);

                        /* check if we need to dispatch a new thread */
                        if (different thread at head of queue) {
                                /* new thread at the head of the queue */
                                DispatchThread();
                        }
                }

                /* send a message to indicate that the update is done */
                SendMsg(SI_UPDATE_DONE, NULL);
        }

        /* deallocate the token */
        Fdealloc(TokenPtr);
}
```

Abort: this is performed when a thread's hard time constraint expires. It pops time constraint blocks until the desired exception level is reached, propagates computation time statistics forward, and deallocates the popped time constraint blocks. In addition, this routine sends message to Application Processor in order to abort other segments of the thread to the desired level. If no time constraint blocks are left, the thread is moved to the importance queue. In any event, a new thread is dispatched when it is necessary.

```
DLAbort() {
        /* adjust the thread control block's state */
        …

        /* it is, so get rid of time constraint stack */
        while (not at desired exception level) {
                /* propagate scheduling information */
                …

                /* deallocate a time constraint block */
                Fdealloc(TCptr);
        }

        /* adjust thread's location in the queues */
        QRemove(DL_Queue, TokenPtr);
        QRemove(IM_Queue, TokenPtr);
        if (no time constraints) {
                QInsert(IM_Queue, TokenPtr);
        }
        else {
                QInsert(DL_Queue, TokenPtr);
        }

        /* check if we need to dispatch a new thread */
        if (different thread at head of queue) {
                /* dispatch new thread at the head of the queue */
                DispatchThread();
        }

        /* send a message to indicate abort */
        SendMsg(ABORT, AbortThread);

        /* deallocate the token */
        Fdealloc(TokenPtr);
}
```

4.5 Future Directions

The Scheduling Facility described in this chapter represents a first attempt at exploring the area of processor scheduling in Alpha. While this work is the first executable manifestation of years of Archons project research in this general area, much of the character of this experiment derives from the underlying testbed's architecture. In particular, the lack of complete, coherent, and symmetrical sharing of primary memory between the Application Processor and the Scheduling Co-Processor resulted in a number of design decisions that are sub-optimal with respect to overall system structure and performance.

Work is currently underway (as a part of Release 2) to rectify the problems introduced by the Release 1 testbed architecture. However, there is still much work that can be done with the existing system, and an effort has begun to perform a series of experiments with Release 1 of

Alpha in order to determine the comparative merits and behaviors of a collection of different scheduling policies. This work involves the creation of a representative real-time application that can impose a realistic load on the system, the implementation of a set of scheduling policies for Alpha, and the generation of a suite of data collection and evaluation utilities.

Given the great computational complexity of the general scheduling problem, the quality of real-time scheduling policies depend heavily on the proper choice of heuristics. Efforts are underway in the Alpha project to refine the exiting scheduling algorithms, develop (and experiment with) new heuristics, and develop optimizations for the scheduling computation's inner loops.

In the future, much of the processor scheduling work in Alpha will focus on extending the results obtained so far in order to better deal with issues of *physical distribution*, *dependencies among schedulable entities*, and *multiple schedulable resources.*

An effort must be made to improve the manner in which the notion time is dealt with across the distributed system. This requires support from the Communication Subsystem, so it is being performed in conjunction with the XTP-based real-time network being used in Release 2 of Alpha. In addition, enhancements must be made in order to deal with the effects that distribution has on the information provided by the application to the scheduler. For example, units of computation with time constraints that are presented to a local scheduler may not be able to properly represent their processing needs, and so false overload conditions may be induced. This is because of the fact that it is possible that not all of a computation will be performed on the local node—a thread may obtain services by executing on other nodes. For this reason, it may be necessary to augment the manner in which the Scheduling Subsystem interacts with the Communication Subsystem (e.g., change it to adjust its loading estimates based on a less pessimistic assumption—as opposed to assuming that all computations will be done locally).

Up to this point in time, all Alpha scheduling work has assumed the complete mutual independence of threads. This is a clearly unrealistic assumption, and therefore work (in the form of a Ph.D. thesis) is underway to extend the current work to cope with scheduling with dependency conditions between threads. No existing scheduling algorithm solves the problem of scheduling a number of activities with dynamic dependency relationships in a way that is suitable for real-time systems. This next unit of work is intended to produce an effective scheduling algorithm, a formal model to facilitate the analytic proof of properties of that algorithm, and simulation results that demonstrate the utility of the algorithm for real-time applications.

Another simplifying assumption made in Release 1 of Alpha is that there is a single Application Processor per node. This made it possible to use scheduling policies based on dynamic deadline scheduling techniques, which do not exhibit optimal behavior when there are multiple processors to schedule. Release 2 of Alpha is being designed to extend the scheduler to work with symmetrical multiprocessors. This involves the modification of the scheduler to manage multiple Application Processors, and to allow the Scheduling Subsystem to execute on any processing element within a node (as opposed to being limited to a specially dedicated scheduling processor).

5 Communication Subsystem

The Alpha Communication Subsystem—that part of the Communication Facility responsible for handling distribution—manages the movement of threads among nodes and guarantees the integrity of the threads as they extend across arbitrary nodes. This latter task includes the detection and elimination of orphaned computations.

Alpha's Communication Subsystem has been designed to provide these required functions by means of a group of communication protocols and to facilitate the development and addition of new protocols. It offers a set of fundamental communication mechanisms upon which a variety of communication protocols may be, and have been, built.

This chapter presents Alpha's communication and distribution requirements, explores Alpha's logical and physical communication architectures, and describes the protocols that have been developed to date to meet Alpha's requirements. The most interesting protocols are discussed in some detail. In addition, this work is placed in perspective with other relevant work.

5.1 Distribution and the Alpha Programming Model

The Alpha programming model has been described in detail in [Northcutt 88b] and an understanding of the basic model is assumed here. However, a few points must be made to explain the effects of distribution within the Alpha programming model. These effects impact Alpha's communication requirements, which are described in the following section.

5.1.1 The Alpha Programming Model

Figure 20 shows a small set of objects and a few threads. Notice that the threads pass through objects and that no physical node boundaries are shown. This is because threads move through objects without regard for node boundaries.

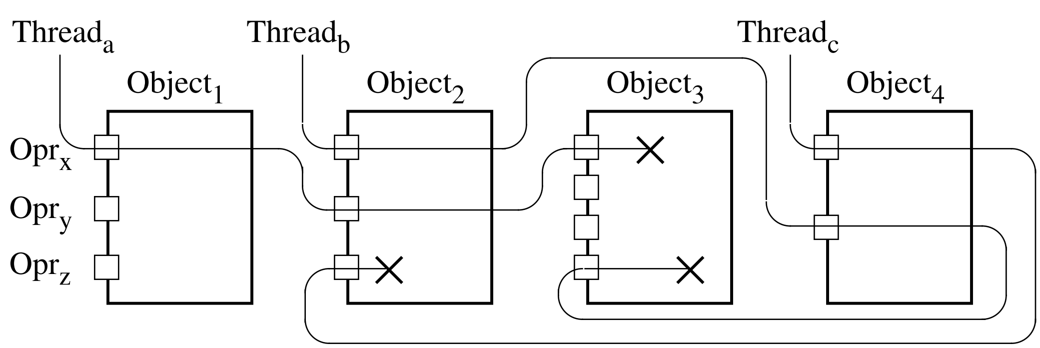


Figure 20: Example Objects and Threads

Every thread has a *root*, and that root is "planted" on the node where the thread was created. Each thread initially executes an invocation on a designated object and may subsequently invoke operations on other objects. Although a thread may span several objects at once (as illustrated by all of the threads in Figure 21), it is only executing instructions in one object at any given time. This active point of control is known as the *head* of the thread, and is indicated by the **X** on each thread in the figure.

A thread can be decomposed into a number of thread *sections*, where each section is the longest, continuous portion of the thread that is contained entirely on a single node. The entire thread can be produced by concatenating all of its sections. For example, in Figure 21, Thread_a has three sections, one on each node.

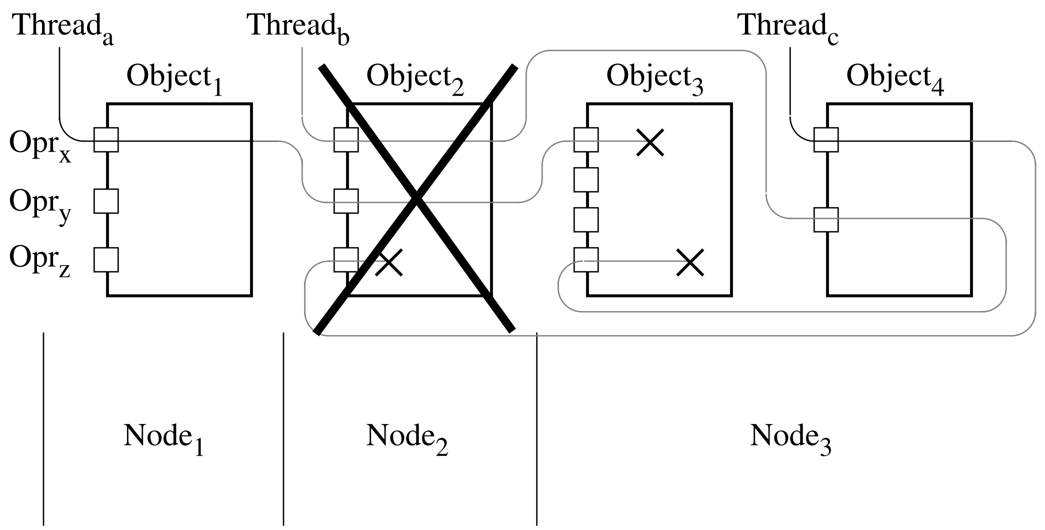


Figure 21: Thread Breaks Due to Node Failure

A thread is *broken* if one or more of its sections are missing due to node or communication failures. Thread breaks often result in orphan computations—computations that can no longer contribute to the overall progress of the thread, and may in fact disrupt its continued progress. To eliminate any potential orphan computations, the thread must be *trimmed* so that the resultant (trimmed) thread has no missing sections between the root and the trimmed head of the thread. The process of trimming the thread is also referred to as *repairing* the thread.

Figure 21 provides examples of thread breaks. The situation shown is identical to that shown in Figure 20. However, in this case, physical node boundaries have been shown; and one node, Node_2, has failed. All three threads have been affected by the node failure and must be repaired. Thread_a, for instance, must be trimmed back to its root section on Node_1, and the orphaned thread section in Object_3 on Node_3 must be eliminated in the process.

Since Alpha is a real-time system, it must execute computations according to application-imposed *time constraints*. A time constraint dictates that a specific group of instructions, possibly spanning several objects, must be executed within a certain interval of time if the program is to behave correctly. Time constraints apply to exception processing—that is, thread trimming due to aborts and failures—as well as to normal processing. Consequently, under normal circumstances, the scheduling parameters for a thread are carried with the thread as it moves through the system making invocations; and under exceptional conditions, scheduling parameters associated with the thread are used to schedule its repair.

At all times a thread is executing under a specific, application-imposed time constraint. In fact, these time constraints may be nested. Whenever a new time constraint is imposed on a thread, the thread executes under the most stringent constraint encountered thus far—it may be the constraint just encountered or it may be the constraint that was already being used for scheduling purposes.

Since Alpha's target applications, and hence their processing requirements, tend to be dynamic and since there are only a finite number of computing resources available, there is a chance that not all of the currently defined time constraints can be satisfied. As a result, some threads may fail to satisfy their time constraints. In that case, these threads are prevented from continuing their normal execution since continuing would mean executing computations at inappropriate times (according to the information provided by the application). Instead, all of the computations that the application desired to complete under the missed time constraint are aborted. Given the nested nature of time constraints, the affected computations are all contiguous, and are all located at or near the thread's head. As pointed out above, the abort processing† for these computations is carried out using the most stringent time constraint for the thread that is still satisfiable. Once the designated computations for the thread have aborted, the thread can continue working to satisfy its next time constraint.

Notice the similarity between the abortion of a computation due to an unsatisfied time constraint and the repair of a thread that is broken as a result of a node or communication link failure. In both cases, a continuous portion of the thread, including the thread's head, must be aborted before the repaired thread may continue normal processing. In fact, the coordination that must be done to carry out the processing is identical in the two cases. The only difference is the reason reported to the application for the abort—a missed time constraint in one case, and a broken thread in the other.

5.1.2 Distribution in Alpha

From a programmer's point of view, Alpha has taken a straightforward approach to handling distribution. By employing the object as the fundamental programming unit and a basic system abstraction, and furthermore declaring that objects are indivisible with respect to nodes—that is, an object cannot be divided between nodes—Alpha allows the programmer to focus on the task of writing consistent and correct objects. Operation invocation on other objects is simple—no attention must be paid (by the programmer) to the target object's physical location. Since the entire object is contained on a single node, the author does not have to worry about failure modes where part of the object's state is unavailable due to failures. Each object is written with a well-defined interface consisting primarily of the operations specified for that object's type. Following a node failure or in response to load imbalances, Alpha may move objects among nodes. And since objects do not share memory, they can be moved independently. Node and communication link failures are manifest as the disappearance of object instances and broken threads. Broken threads are repaired automatically, leaving the programmer to handle the problem of replacing or relocating object instances. This is typically accomplished by creating new instances to replace old ones, or by recovering or migrating old instances.

To simplify the programmer's task and to reduce the bookkeeping required of Alpha, invocations are performed in a location-independent manner—that is, the *invoker* does not need to know the node on which the *invokee* (the target object specified in the invocation) is located. As a result, Alpha must be able to find the invokee for any given invocation dynamically.

---

†*Abort processing in a real-time system is not quite the same as abort processing in other systems, such as transaction processing systems. A real-time computer system typically affects a physical process as part of its normal operation. As a result, abort processing must include not only the restoration of computer memory state, for example, but potentially must also include compensation for actions taken that affected the physical process.*

As was pointed out above, distribution complicates the maintenance of thread integrity (making certain that a thread is entirely intact at all times) and introduces the notion of thread repair. It also demands that Alpha propagate time constraint information along with threads as they move among nodes. When a thread must be repaired due to either a thread break or an abort due to an unsatisfied time constraint (or an aborted transaction), the time constraints for the thread must be circulated in order to schedule the repair correctly. In addition, the affected portion of the thread may span several nodes, so that the abortion of each affected section must be initiated and tracked across the Ethernet.

Not surprisingly, decentralized thread repair is more complicated than centralized thread repair, where centralized thread repair is the thread repair function provided on a single-node system. On a uniprocessor, if the processor fails, all computation stops. On a multiprocessor, if a processor fails, the remaining processors can note that fact and act accordingly, using shared memory to instantaneously and consistently communicate information with one another. In a multi-node system, the loss of a processor or node can be detected by other nodes independently. There is no shared memory to provide a consistent system-wide view for all of the nodes. Rather, each node has its own view of the overall state of the system. Following a node failure, all of the surviving nodes that are affected must detect the failure and coordinate their recovery actions. During the course of handling one failure, another can occur, potentially complicating matters. Although clustered failures may seem unlikely, they may in fact occur often—even during normal operation. For instance, while trimming a thread due to an unsatisfied time constraint, another section of the same thread could be lost due to a node failure, or a second time constraint for the same thread could be missed, necessitating more abort processing (thread trimming).

In addition to timeliness, Alpha is concerned with reliability. In a reliable system, critical data is highly available—that is, despite node and communication link failures, it can be accessed in short order. One way to increase the availability of data in a distributed system is to replicate it. Towards that end, Alpha provides a set of mechanisms that support various replication policies. The major work on the development of replication policies for real-time systems has not yet begun, but a simple initial policy has been put in place in order to begin defining and exercising the necessary replication support mechanisms.

In Alpha, any object can be replicated, and there is no special code inserted in the object to provide replication support. Instead, all of the support is provided by the operation invocation facility and the object naming facility. There are a number of copies of a replicated object, and one of them is designated as the master copy. The others are backup copies. All invocations on the replicated object are actually performed on the master copy, and the changes made by each operation are propagated to the backup copies. Should the master copy ever be lost, a simple procedure is employed to promote one of the backup copies to be the new master. Optionally, additional backup copies can also be created. Furthermore, threads are placed in the backup copies by the ***ObjectManager*** on the master copy's node in order to detect their failure. Should a backup copy ever disappear, the thread that was placed in it will break, and the thread will be trimmed—allowing the thread to execute code in the ***ObjectManager*** again. This code replaces the backup copy, if possible. Consequently, the ***ObjectManager*** makes sure that there are always backup copies available in case the master copy should fail.

The Alpha programming model also includes the use of atomic transactions to aid in the construction of reliable applications. The abortion—and, in fact, the completion—of a transaction will require similar coordination to that involved in trimming a thread due to an unsatisfied time constraint. In the case of a transaction, all of the objects participating in the transaction will be involved—even those that no longer host a portion of the thread executing the transaction. As with replication, much of the advanced work related to the development of atomic transactions for real-time systems remains to be done. However, the general demands that transaction support will place on the Communication Facility can already be anticipated.

5.1.3 Related Work

Like Argus ([Liskov 84]), Clouds ([McKendry 84]), Eden ([Almes 85]), CRONUS ([Schantz 85]) and other similar operating systems, Alpha explicitly addresses issues of physical distribution. In addition, all of these systems use the same general abstraction—that of an object—as the basic building block of an application. All but CRONUS feature the use of transactions in order to support the construction of robust applications, as does Alpha. Argus has explored the use of object replication ([Herlihy 84]) as well. Although Alpha has provided mechanisms for transactions and object replication, most of the development of the policies governing their use in a distributed real-time system has yet to be done. CRONUS has emphasized the program development environment and object management issues, areas that have received mush less attention in Alpha.

Alpha is distinguished from these systems on two counts. First of all, Alpha explicitly addresses real-time applications. None of the other systems is directly attacking this issue. Timeliness considerations pervade Alpha, and as a result, even functions that are apparently the same in all of these systems—for instance, orphan detection and elimination—differ to a certain extent.

Secondly, all of the other systems—except Clouds—have points of control that are held captive within a single object. In Alpha, these points of control are free to move among objects. This movement among objects can be accomplished in the other systems by means of remote procedure calls ([Nelson 81]) or message passing. The key difference is that Alpha's threads move through the system by themselves—always carrying their own thread-specific scheduling parameters. This scheduling information is not propagated in the other systems, due to the fact that they do not focus on real-time issues. By always propagating the thread's scheduling parameters with the thread, Alpha allows the application writer to create threads that correspond to sequences of activity in the physical world and be assured that the time constraints imposed by the physical world will be considered at all times as the thread moves throughout the system. In the other systems, intermediate server objects with their captive points of control must be used and will have scheduling parameters that are unrelated to the point of control on whose behalf they are executing.

Clouds comes closer to Alpha in this respect than do the others. It has processes that can move among objects. Its processes are actually equivalent to objects that have a thread embedded in them when they are created. In essence, Clouds has two distinct types of objects—passive and active; while Alpha has only one type, passive, and activity is generated by an entirely different entity—the thread.

Many typical operating systems—Mach ([Accetta 86]) and UNIX ([Ritchie 74]), for example—manage only a single node. Communication with other nodes is carried out by system, or even user, utilities. This contrasts sharply with the approach taken in Alpha where communication, in the form of operation invocation with thread maintenance and repair, lies at the very heart of the system. Each node is designed to behave as a single part of a larger whole, and it is aware that there are other nodes present and that coordinated effort is required to support useful applications.

5.2 Communication Requirements

Alpha has three distinct types of communication requirements. First of all, there are the functional requirements—that is, the set of communication functions that must be performed in order to support the Alpha programming model, as outlined above. Secondly, the experimental nature of Alpha imposes flexibility and adaptability requirements. Finally, the IMI interface and the Alpha hardware architecture constrain communication approaches.

The Alpha programming model dictates the functional requirements for Alpha's communications:

1. implement a location-transparent operation invocation function that propagates real-time scheduling information along with the invoking thread;
2. monitor the condition of active threads — detect thread breaks and repair broken threads as they are detected;
3. support the simple replication scheme described above;
4. provide communication between Alpha threads and objects and UNIX processes — this capability is needed since Alpha, as a research operating system, does not have all of the program development and monitoring facilities, among other things, that UNIX does.

Alpha is a research operating system, and hence its long-term requirements are not precisely known. Since it is a distributed system where communication functions are assumed to be the backbone of the system, Alpha's Communication Facility must be flexible in order to support the changing needs of the system. In addition, Alpha is intended to act as a testbed platform for experimentation with various implementations of a single operating system function or the interaction of designated functions. Once again, the Communication Facility must be flexible in order to meet these requirements.

5.3 Alpha Communication Architecture

The need for a flexible Communication Facility that could be used in a testbed dominated the determination of the Alpha communication architecture.

The functions that must be provided by the Communication Subsystem are quite diverse, and so an approach was taken that employed a number of special-purpose communication protocols, rather than a single general purpose protocol that could support all of the required functions.

Although the protocols were quite different—some were point-to-point, others were broadcast or multicast†, and yet others used a mixture of these addressing modes; some dealt with only a single invocation, others dealt with many invocations; some were executed periodically, others were not; some used fixed length messages, others used variable length messages—they still demanded some common lower-level support to handle message transmission and receipt, message buffer management and manipulation, time management, and other similar concerns.

---

†*In a broadcast, a message is sent to all receivers in the system; in a multicast, a message is potentially sent to more than one, but (perhaps) fewer than all, receivers, and those that are not receivers are unaffected by the message. Both broadcast and multicast messages represent instances of one-to-many communication.*

To provide this common low-level support, while allowing the development of a number of independent protocols, the resultant architecture (see Figure 22) is layered, providing a set of mechanisms that may be used as primitives by any number of communication protocols. The architecture consists of two key components:

- the Communication Virtual Machine;
- communication protocols.

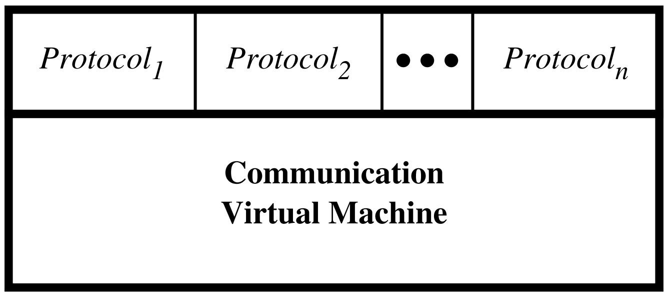


Figure 22: Logical View of the Alpha Communication Architecture

The next two sections describe the Communication Virtual Machine and offer a general discussion of communication protocols within the Alpha communication architecture. Later, the specific protocols that have been implemented to meet Alpha's communication requirements will also be examined.

5.3.1 The Communication Virtual Machine

The Communication Virtual Machine (CVM) is a simple executive† running on the CP, augmented by the communication mechanisms. It allocates the single processor among all of the protocols that desire to use it in a first-come-first-served (FCFS) manner.

As shown in Figure 23, the fundamental data structure within the CVM is the *TokenQueue*. All of the work that must be done is represented in the CVM by tokens. Each token (see Figure 24) has a destination field that indicates the protocol handler for which that token is intended. In addition, each token has fields indicating the source (creator) of the token and its protocol-

† *An executive is a limited operating system. In this case, it is responsible for scheduling the processor and handling interrupts.*

specific type. As work progresses or arrives, new tokens are added to the *TokenQueue*. As work is dispatched, tokens are removed from the queue.

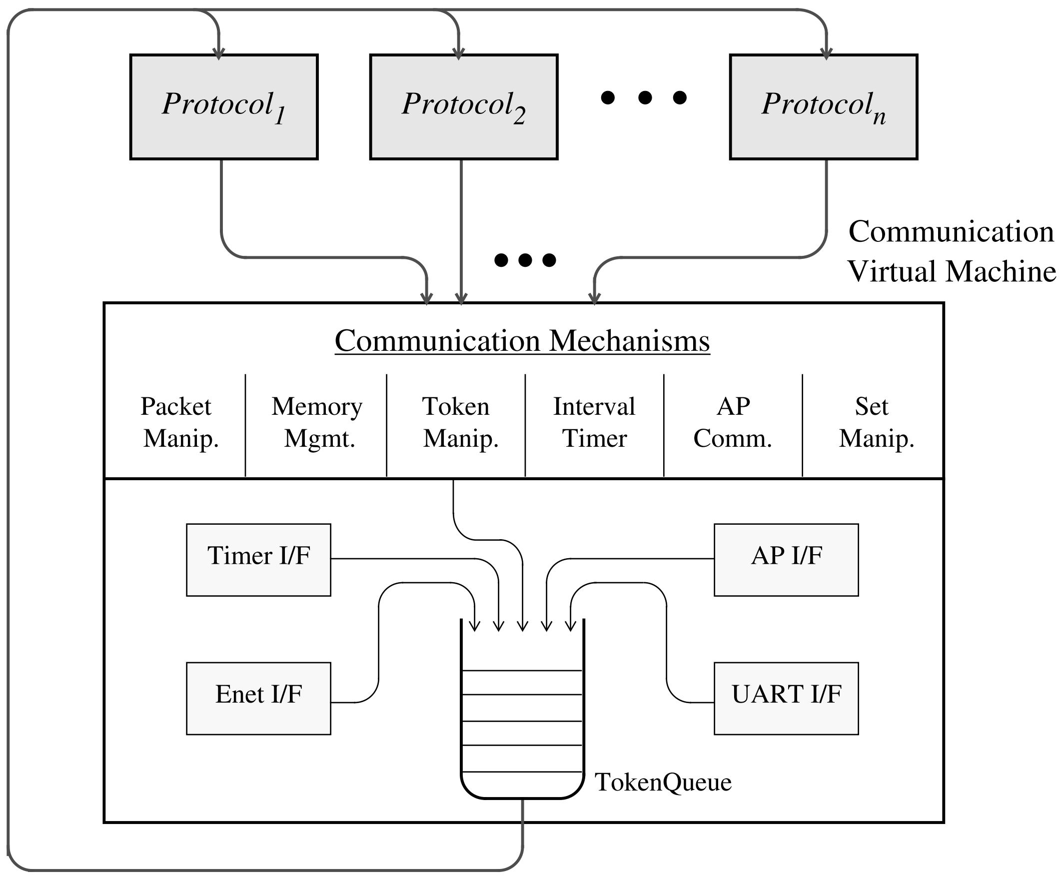


Figure 23: Expanded View of the Communication Architecture

Tokens are added to the *TokenQueue* in response to certain specific *events*. These events may originate from any of a number of physical devices—the Applications Processor, the Ethernet controller, the serial input/output (I/O) controller, or the interval timer—and include:

- arrival of tokens from the Application Processor Interface;
- arrival or departure of packets from the Ethernet interface;
- arrival or completion of I/O from the serial interface;
- expiration of an interval timer;
- occurrence of specific exception conditions.

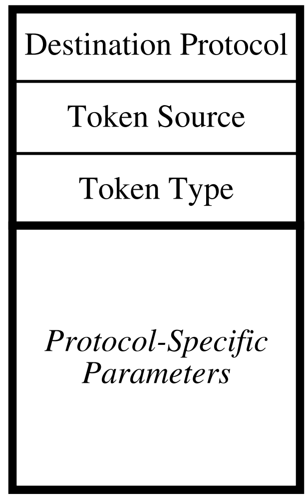


Figure 24: Communication Virtual Machine Token Format

Furthermore, one protocol can generate tokens destined for another protocol, thus allowing the construction of hierarchical protocols. This allows one protocol to use the services provided by others and also facilitates implementations of higher-level protocols that are encapsulated by lower-level protocols.

The CVM employs a simple scheduling algorithm—it simply removes the token from the head of the *TokenQueue* and schedules the protocol handler named in the Destination Protocol field of the token. That token handler is then allowed to execute until it has finished processing the token, at which point it surrenders control of the processor to the CVM executive once again.

Unless they are explicitly disabled, interrupts can occur at any time and will temporarily seize control of the processor. In every case, the interrupt handlers execute for the briefest time possible—they gather whatever state must be recorded in order to perform the bulk of the interrupt handling at a later time and place a token in the *TokenQueue* to initiate that processing.

Protocol writers use a set of communication mechanisms supplied by the CVM as fundamental building blocks for constructing their protocols. These mechanisms include:

- Ethernet packet-handling mechanisms — to create and delete packets; send and receive packets; encapsulate and merge packets;
- interval timer mechanisms — to schedule and cancel timed events;
- memory management mechanisms — to allocate and deallocate shared Multibus memory pages;
- Application Processor interface mechanisms — to send commands to the AP; receive commands from the AP;
- set manipulation mechanisms — to create and delete sets; lookup, add, and remove elements in a set.

5.3.2 Communication Protocols

Protocol definition consists of composing the communication mechanisms to implement the desired protocol. Most of the protocols composed to date consist of a set of relatively short *actions* that are executed at critical times. These actions take the form of routines that are ex-

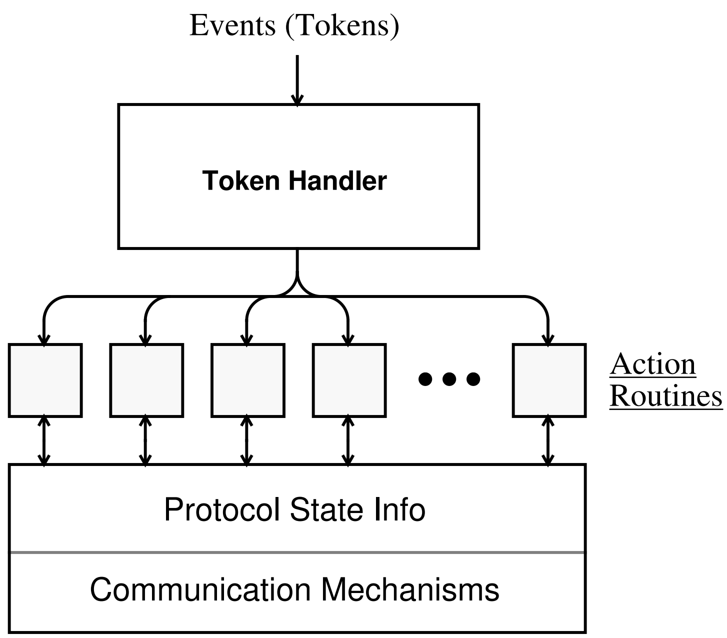


Figure 25: Logical Protocol Structure

ecuted in response to the initiating event. To date, these actions have been quite short in terms of the number of steps required to handle the event, which is an encouraging sign. This means that any event for any protocol can be processed fairly quickly, therefore allowing high event-handling throughput to be obtained for the CVM.

Protocols use the CVM to do only what *must* be done at any given time. As a result, they do not spin and wait for another event to occur. Rather, they define an action to be taken in response to that later event and release the CVM for the time being. Of course, many, if not all, protocols will also set a watchdog timer (using the interval timer mechanisms) to guarantee that no work is lost in case the later event never occurs.

Figure 25 shows the logical structure of a protocol. All of the events associated with that protocol are embodied by CVM tokens and are routed to the protocol's *token handler*. Based on the token's source and type, the token handler selects the *action routine* to execute to handle the event. Each action routine has access to the protocol's private state information and to the communication mechanisms provided by the CVM.

To specify a protocol, the protocol writer must determine the set of events that may occur during the execution of the protocol, write a token handler that will identify each of them, and specify the steps to be taken by the corresponding action routines for each of them. The flow of control and the interaction of events can often be visualized conveniently in a graphical form, such as the state diagram shown in Figure 26.

The protocols described in this document were all written by hand—that is, they were written in C, without the benefit of a special-purpose, protocol definition language. However, they were written in a very stylized manner in order to explore the feasibility of defining and developing such a language. In the future, a protocol definition language, along with other tools—such as a protocol analyzer—might greatly simplify the design and development of protocols.

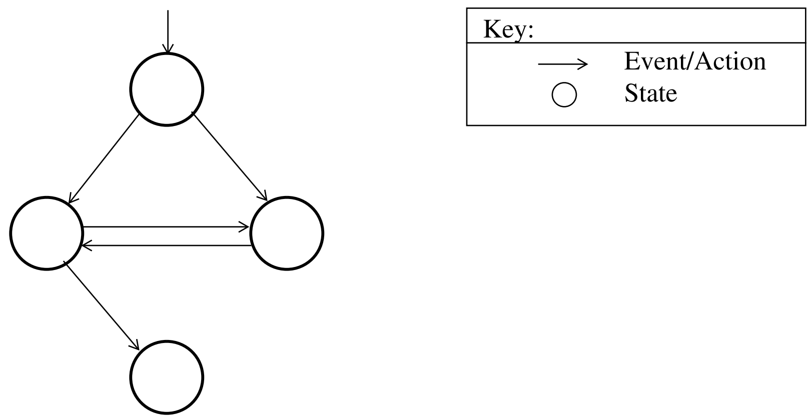


Figure 26: Protocol State Diagram

5.3.3 Related Work

The Alpha Communication Architecture employs a dedicated processor to provide hardware concurrency in processing and to increase the performance of communication tasks. The use of dedicated hardware for these purposes is by no means new. Similar hardware support dates back at least to the 1960s ([Thornton 64], [Jensen 78]), and continues to the present.

Recently, a number of commercial products have provided hardware communication support with a new twist—the products are communication engines that run specific standard protocols, such as TCP/IP, for specific purposes (like the Micom-Interlan Ethernet NPx00 Network Processor series of protocol processor boards). This is similar to the approach taken in this work, except that, for Alpha, it was deemed important to be flexible enough to support research into a number of special-purpose protocols, not to simply run standard protocols. In fact, some products do support user-supplied protocols ([CMC 85] and the Interphase V-Ethernet 4207 Eagle, among others), and substantial protocol support, much like that desired by Alpha, is being tackled by a commercial venture as well ([Chesson 87]).

There is some debate as to whether it is better to employ general-purpose, standard protocols or to use special-purpose protocols. Typically, the advocates of standard protocols argue that there is little loss in performance introduced by using the standard protocols; there is, almost by definition, wider applicability for applications that use standard protocols; and there is far less time and effort spent on developing redundant functions in a number of different protocols, as might be required for special-purpose protocols ([Sproull 78]). On the other hand, standard protocols do not necessarily provide adequate support for some functions that may be desirable in scalable, distributed systems, such as multicast capabilities. Since Alpha must be free to explore the use of such features, it has chosen to pursue special-purpose protocol development. At the same time, by allowing the construction of hierarchical protocols ([Fleisch 81]), Alpha removes the need for protocols to reimplement functions already available from other protocols.

A considerable number of models for protocol specification have been developed. Some of these take a pragmatic approach to define languages and primitives for communication ([ISO 86], [CCITT 85] and [Nash 83]). Others ([Bochmann 80a], [Gouda 85]) are more interested in defining protocols within a formal model to facilitate verification of correctness and demonstration of properties.

There are many models of protocol specification based on finite state machines (for example, [Bochmann 82] and [Brand 83]), and often protocols are represented graphically in this work. Many of these models differ in details, but the general consensus seems to be that this is a reasonable expressive model for the task at hand. Some finite state models, like the one used for Alpha, do not support non-determinism. That is, they impose a total ordering on events such that, at any time, from any state, only one state transition's precondition is satisfied and, therefore, only one transition is allowed. Other models ([ISO 86]) allow a form of non-determinism where the preconditions of multiple state transitions may be satisfied at once, but still only one transition is chosen (non-deterministically) from among the candidates. In the future, Alpha, inspired by the desire to explore state exchange protocols ([Fletcher 79]) and to develop more general methods to think about protocol specification and behavior, may wish to explore a finite state model that pushes non-determinism one step further—in effect, if several state transition preconditions are satisfied at one instant, then all of them are taken. This is reminiscent of a fork operation and has been used in general finite state machine models before ([Hopcroft 79]). Some of these forked points of control would later cease to exist if no state transition preconditions were satisfied at a later point, while other control points would continue.

The actual development of the Alpha protocols and the CVM mechanisms applied a perspective partially gained through previous work in the area. General design principles of protocols are covered in [Danthine 81]. The notion of using structured building blocks— mechanisms and other protocols—for protocols was influenced by earlier work (such as the Datakit approach outlined in [Chesson 80]). Finally, the CVM's mechanisms were defined and implemented using optimization techniques, such as those in [Clark 82].

5.4 Alpha Communication Protocols

To meet the functional requirements of the Communication Subsystem, six protocols have been defined. They are:

- **Reliable Message (RM) protocol** — sends packets reliably;
- **Remote Invoke (RI) protocol** — implements remote procedure call (RPC) semantics for invocations whose targets are on another node;
- **Thread Maintenance and Repair (TMAR) protocol** — maintains thread integrity and coordinates repair of broken and aborting threads;
- **Page Transfer (PT) protocol** — transfers memory pages between object instances;
- **Monitor Active Nodes (MAN) protocol** — monitors the "health" of all of the other nodes in the system;
- **Alpha/UNIX Link (AUL) protocol** — provides communication with UNIX systems or applications.

Most of these protocols are quite straightforward and so will be described very briefly. TMAR is the most complex, and the most Alpha-specific, and will be discussed in more depth, as will the RI protocol.

5.4.1 RM Protocol

The RM protocol is quite straightforward. It sends a packet reliably to a single Ethernet destination address†. The sender transmits the packet and waits for an explicit acknowledgment from the receiver. The sender sets a watchdog timer in case an acknowledgment is not received within the expected time. If this timer ever goes off, the packet is resent and another watchdog timer is set. This process continues until either an acknowledgment is received or a programmer-specified maximum number of retries has occurred.

The sender portion of the RM protocol always returns a result to its requester—a successful result indicates that an acknowledgment was received in a reasonable amount of time following a transmission of the packet; a failure result indicates that this did not happen. Notice that the protocol could indicate a failure even though the receiver actually got the packet—e.g., due to a heavy load on the remote node, the acknowledgment could arrive late; or a network failure could prevent the successful transmission of an acknowledgment.

The receiver portion of the RM protocol is able to recognize the duplicate packets that may occasionally be sent by keeping track of the sequence numbers for all of the recently received packets from each node in the system. These duplicates are discarded.

The design and implementation of the RM protocol follows the above description nicely. It has four events and four corresponding action routines. The events are:

- the arrival of a request to send a reliable message;
- the arrival of a reliable message;
- the arrival of an acknowledgment for a reliable message;
- the expiration of a watchdog timer.

Although the RM protocol provides a useful communication capability by itself, it is primarily used as a building block for other protocols.

5.4.2 RI Protocol

The RI protocol implements the remote invocation function for Alpha. Operation invocation has essentially *remote procedure call* semantics. In addition, since threads may break at any time, potentially interacting with outstanding invocations, there is a fair amount of interaction between this protocol and the TMAR protocol to be discussed below.

Since each node in the testbed system is a multiprocessor, there are a number of participants involved in carrying out a remote invocation. In particular, the Application Processor, the Communication Co-Processor, and the Scheduling Co-Processor all play a role in remote invocation.

Prior to an invocation, a thread executes code in some object on the invoking node's AP. At some point, it encounters an operation invocation to be performed on an object that is not located on its current node. The Kernel Proper determines that the invocation is directed to an object instance that is not local. The kernel then assumes that it must be intended for a remote object and prepares to perform a remote invocation. To do this, it acquires the scheduling parameters for the invoking thread from the SP, places these parameters with the invocation's request parameters, manipulates memory mapping tables to allow this information to be accessed from the Multibus backplane, and notifies the Communication Subsystem on the CP to perform the remote invocation. At that point, the AP blocks the invoking thread section until it has received the reply for the invocation from the CP. Later, after receiving the reply from the CP, the kernel takes the updated scheduling parameters, which are included with the invocation's reply parameters, and sends them to the SP so that scheduling decisions are made using the most up-to-date information. The invoking thread section is then unblocked and is executed whenever the SP dispatches it.

On the AP of the invoked node, the kernel is notified by the CP that an invocation has been received from another node and is given access to the invocation parameters and the scheduling parameters for the invoking thread, both of which are stored in shared Multibus memory. The AP then passes the scheduling parameters to the SP and schedules a new thread section to be executed using the thread's scheduling parameters. At a time determined by the load of the node and the scheduling parameters under which it is executing, the new thread section will be dispatched. It will execute, possibly performing other invocations, until it completes the invoked operation, at which time it initiates the reply sequence on the AP. This causes the AP to acquire the updated scheduling parameters for the thread from the SP, include those scheduling parameters with the invocation's reply parameters, deallocate the new thread section and notify the CP that the invoked operation has completed.

The CPs on the invoking and invoked nodes coordinate the AP actions just described by means of the RI protocol. The protocol itself has four distinct phases, two on the invoking side and two on the receiving side, that interact with the AP actions just described at the obvious times:

- request leaves the invoking node;
- request arrives at the invokee's node;
- reply leaves the invokee's node;
- reply arrives at the invoker's node.

For each of these steps the RM protocol is used to provide a reliable communication path between the nodes involved.

When an operation invocation is initiated on the invoking node, the invoking CP encapsulates the invocation request parameters and scheduling parameters in a message, addresses the message to the invoked object, and uses the RM protocol to reliably send the message to RI protocol on the node containing the invoked object. A record of the ongoing invocation is created for bookkeeping purposes. After the attempt to reliably send the invocation request, the CP waits to receive a thread synchronization message from the invoked node. This marks the point at which the head of the thread has "officially" moved from the invoking node to the invoked node.

On the invoked node, the RI protocol receives the message requesting the invocation and copies the invocation parameters and scheduling parameters into shared Multibus memory. The AP is notified that the invocation has been received and given the address in the Multibus address space of the parameters. The RI protocol coordinates with the TMAR protocol to register the new section of the invoking thread and then sends the thread synchronization message to the invoking node.†

The analogous steps are taken during the reply sequence on the two nodes. Once again, the major complication is making an orderly transition in passing the point of control—that is, the head of the thread—from one node to the other. And in every case, there is always the problem that the thread may have broken and care must be taken to maintain consistent records and to reclaim resources (such as Multibus memory pages) that belong to trimmed threads. This is a theme that occurs again and again in the design of the more complicated protocols: points in the protocols must be identified where asynchronous conditions are resolved. For example, the CVM framework provides an environment in which every action routine is executed atomically, and since events are manifest in the CVM by tokens that are queued in the *TokenQueue*, the queue acts to serialize events. The protocol designer must take advantage of these facts in creating intricate protocols in order to reduce the complexity of the task. The task is already difficult in that much of the work results from the need to coordinate two or more physically dispersed CPs that are truly executing concurrently. Complicated interactions can result.

To permit a form of logical addressing to be used, the Communication Subsystem maintains a list of the logical names it must recognize. This list is examined each time a logically addressed packet arrives to determine whether the node should accept the packet. Currently, this list is implemented as a hash table whose contents are derived from the Kernel Proper's Dictionary and represent the objects and threads that exist on the node, as well as, other logical entities local to the node (e.g., transactions). When the kernel adds entries to, or removes entries from, the Dictionary, the Communication Subsystem is notified (via an IMI command), in order to keep its logical name table up-to-date.

5.4.3 TMAR Protocol

The TMAR protocol monitors the integrity of threads and repairs threads when a break in the thread is detected. It is also responsible for coordinating the trimming of a thread due to an abort—for instance, due to an unsatisfied time constraint. It is an excellent example of a decentralized protocol executed by a number of identical peers asynchronously on each node in the system.

Every thread has a root, and this root is located on the node containing the object in which the thread was created. Each node is referred to as the *root node* for all of the threads that are rooted on that node. It is the responsibility of the root node to maintain the integrity of all of the threads rooted locally.

To accomplish this, a three phase sequence is executed periodically by each root node. In the first phase, the root node broadcasts a polling message, containing a list of all of the threads rooted at the root node that have made remote invocations, and requesting that any nodes con-

---

†*There is a separate thread synchronization message so that there will always be at least one node responsible for reporting the presence of the thread's head section. In fact, there is a short interval of time during each invocation when both the invoker's node and the invokee's node will report the presence of the thread's head. If, instead, no separate thread synchronization message were sent, then the responsibility for reporting the presence of the thread's head would be implicitly passed from the invoker's node to the invokee's node with the initial invocation packet. As a result, during the time that the packet was in transit and being interpreted by the invokee's node, no node would report the presence of the thread's head. And failure to report the thread's head, under certain circumstances, can result in errantly trimming a section from an unbroken thread. The discussion of the TMAR protocol should make all of this clearer.*

taining sections of these threads report this fact, along with information related to the location of the thread's head, to the root node. In the second phase, each of the nodes containing sections of the listed threads return the requested information to the polling node. This information is examined by the polling node to verify that an unbroken sequence of sections exists between the root and head of each thread in the list. (Some missing section and head reports are tolerated, since communications over the Ethernet are subject to errors. However, a consistent missing section or thread head is certainly a problem.) In the third phase, the polling node broadcasts a refresh message that announces the health of the threads that are rooted locally to all of the other nodes. If a thread is unbroken, the message indicates that the entire thread should be refreshed; if it has broken, only the continuous portion of the thread that begins at the root and ends at the point of the break is refreshed. The sections beyond the break must be eliminated (aborted) before the thread may resume execution at the point of the break. Upon receipt of a refresh message, the other nodes refresh local thread sections as directed.

If a specified time interval is exceeded since the last refresh message for the sections of a thread on a node, then it is assumed that the root node for that thread has failed. In that case, all of the surviving sections are orphans and must be eliminated.

Over time, this three phase poll-respond-refresh sequence keeps all of the unbroken threads intact, without bothering the AP.

In addition, due to the use of broadcast and multicast messages, the amount of information that must be maintained for each thread in the system is small. For instance, the root node never necessarily knows where the head of a given thread is, or in fact, how many sections the thread currently has. It cannot be sure of such information because it only knows what it received in response to its most recent polling message. Since messages may be lost, it cannot be sure that it received a report about all of the existing sections. Furthermore, even if it did receive all of the responses, the thread may have subsequently executed a new invocation that took it to another node, thereby creating a new thread section.

There is no need for the root node to be aware of such invocations. It is only attempting to ensure thread integrity, and that can be accomplished by detecting thread breaks, which are characterized by missing thread sections, possibly including the head section. Threads are free to move freely without explicitly reporting their movements to the root node as long as they do respond to the root node's polling messages.

Polling messages are sent periodically by each node in the system. Also, the time limit within which a thread must be refreshed is a programmable, integral number of polling cycles, as is the number of responses for any thread that must be missed before it is assumed to be broken. Therefore, the polling frequency establishes the responsiveness of the system—that is, the length of time that passes after a thread break occurs until it is conclusively detected. This allows the protocol to trade communications bandwidth for thread break responsiveness: if necessary, a system can be made more responsive by utilizing adequate bandwidth to use a sufficiently short polling interval to satisfy any responsiveness requirements.

The Alpha programming model dictates that threads must be trimmed according to the proper application-defined scheduling parameters for the section that will be at the head of the trimmed thread. The TMAR protocol handles the circulation of the proper scheduling parameters for trimming a thread.

As a thread moves from one node to another making an invocation, it carries with it only the scheduling parameters under which it is currently executing. No information concerning any other previously established, looser time constraints moves with the thread.† Instead, whenever other scheduling parameters are needed to schedule thread section aborts during a thread repair, TMAR locates the proper parameters and circulates them to the nodes that need them.

In the case of a thread break, the proper scheduling parameters are known by the node having the thread section immediately preceding the first break in the thread (that is, the section adjacent to the missing section but nearer to the thread's root). This section will be the head section of the repaired thread. When the root node for the broken thread determines that a break has occurred, it determines the section identifier of the new head section and sends a refresh message that only refreshes the thread up to and including that new head section. All sections beyond that point will be marked for abortion following receipt of the refresh message. If the new head section happens to be located on the thread's root node, then the scheduling parameters under which the abort should be performed are included in the refresh message. Otherwise, when the node holding the new head section receives the refresh message, it notices that it holds that section and sends a message directly to the root node containing the desired scheduling parameters. The root node includes these parameters in all subsequent refresh messages for the thread until the thread repair is complete. Until the proper scheduling parameters are received, the abortion is scheduled on each node using a default time constraint that indicates the abort must be completed in the distant future (literally, by the end of time as far as the system is concerned). In this way, progress can be made on the abort if there is no other work to be done in the meantime; and if there is work to be done, the abort will not interfere with it until the proper scheduling parameters are received.

The situation is somewhat simpler when TMAR is trimming a thread due to an abort, such as an unsatisfied time constraint. In this case, the node on which the unsatisfied time constraint was initially established detects the fact that it has been missed, and immediately notifies the root node of this fast along with the desired scheduling parameters that are to be used to schedule the abort processing. As before, the root node then circulates these parameters to the other nodes containing sections of the thread being trimmed.†

During the trimming of a thread, each node trims the sections that are local to it that are beyond the point of the break or abort. After all of these sections have been aborted, the root node for the thread is notified of that fact. The root node continues to perform its normal polling

†*This is done to limit the amount of scheduling information that must be propagated with a thread. Since time constraints can be nested arbitrarily deeply, there is no fixed bound that will necessarily hold all of the scheduling information for a thread. Therefore, it cannot be assumed that the scheduling parameters that are needed following an abort or thread break can always be carried with the thread. Consequently, there must be some method to locate the necessary parameters if they are not among those that are propagated with the thread. Since thread breaks will occur at arbitrary locations within a thread, and since there is no required correlation between the tightness of time constraints and the point (with respect to the thread's head) at which they were defined, it makes sense to minimize the amount of scheduling information propagated with the thread, and to use the method that must be provided in any case to locate and circulate the proper scheduling parameters to be used during thread trimming.*

†*Actually, each node having a section of the thread operating under the unsatisfied time constraint will independently detect the fact that it has not been met. However, only the node on which the time constraint was established knows the time constraint that was superseded. However, by detecting it themselves, the other nodes have the opportunity to carry out some, or all, of the abort processing (under default scheduling parameters) without having to wait for notification from another node.*

cycle for the thread, refreshing only that portion preceding the break or abort point. It keeps track of the aborted sections as it is notified of them and determines when the entire thread trim operation has been completed—that is, when each section beyond the break or abort point has either been aborted or determined to have failed (presumably because of a node or communication link failure).

Once the entire thread has been trimmed, a notification is sent by the thread's root node to the node containing the trimmed thread's head signalling that the trimmed thread may resume execution.

Since TMAR on a thread's root node continues to perform its normal three phase refresh function even while the thread is being trimmed, the protocol can handle successive failures. That is, while a thread is being trimmed, subsequent thread breaks or aborts will be handled as soon as they are recognized. This makes the protocol somewhat more complicated than it would have to be if subsequent breaks and aborts could be deferred, but it will improve responsiveness and reduce the time that the trimmed thread must wait before proceeding in several cases, most notably when multiple nested time constraints cover a relatively small interval of time.

Although it is straightforward for the root node to wait for the next polling cycle to take many actions, responsiveness can be enhanced if it acts as soon as possible. Consequently, optimizations have been inserted into the protocol to allow the root node, and in some cases other nodes, to act without waiting for a polling cycle.

The root node's task of determining the state of a given thread is not as simple as it might seem. This is because of the asynchrony among nodes and, hence, TMAR instances. When the root node broadcasts it polling message, the thread is at a single definite place. Each node that is aware of the thread responds to the polling message, but only after processing it, which involves executing a TMAR action routine to determine which threads are of interest. Since all of the nodes have *TokenQueues* of different length and operate asynchronously, their TMAR action routines will execute at different times and, as a result, their views of where the thread's sections are may differ. For example, if the thread is performing an invocation, possibly two different nodes will report having the head of the thread. Added to the possibility that either the polling message or a node's response may be lost due to the nature of the Ethernet, this makes it difficult to be sure about the condition of the thread based on polling information.

5.4.4 PT Protocol

The PT protocol was designed as a general utility, but there was a specific need for it as well—supporting a primitive version of object replication. In the future, it may be employed by the Secondary Storage Subsystem, a debugger or monitoring system, or the invocation facility for large parameters.

The PT protocol transfers arbitrary pages from one object to one or more other objects. The RM protocol is used to send these pages reliably to each destination. Unfortunately, due to Ethernet packet size restrictions, two packets are required to hold a page of data. This means that each transfer requires two individual steps, which complicates coordination somewhat when compared to a single step. (Remember, failures can happen at any time on either end of the transfer.) Of course, the additional coordination and the number of packets required act to decrease the performance of the protocol.

5.4.5 MAN Protocol

The MAN protocol, like the PT protocol, was included in the initial suite of protocols in order to support Alpha's first replication scheme. This protocol allows each node in the system to keep a fairly accurate picture of the status of all of the other nodes in the system at any given time. Once again, the protocol is decentralized, and so each node will have its own view of the other nodes' status.

Each node periodically broadcasts a packet announcing its presence to the other nodes in the system. When such a packet arrives from another node, it indicates that the node should be assumed to be active. If it had previously been assumed to be inactive, then this constitutes a change in state, and the kernel is notified of the change.

If any node has not been heard from for a sufficiently long time, then it is assumed to be inactive. Once again, if this constitutes a change in its assumed state, the kernel is notified of the change.

Finally, the kernel may request a complete report of the current node status information, and the CP will oblige.

5.4.6 AUL Protocol

The AUL protocol provides a link between Alpha and a UNIX environment. Specifically, it allows reliable messages to be sent between the two environments. The reliability is provided by means of positive acknowledgment of messages and the detection and suppression of duplicate messages. Given the UNIX development environment under which Alpha was developed and initially tested, a facility to link them together seems quite advantageous.

In the UNIX environment, AUL presents a standard UNIX interprocess communication (IPC) interface to the Alpha environment. That is, stream sockets, based on DARPA† standard TCP/IP ([Postel 81a], [Postel 81b]), provide the connection to Alpha, allowing flexibility and ease of use for the UNIX programmer. Communications from Alpha are also presented to UNIX processes through stream sockets.

In the Alpha system, an object interface is provided for communication to UNIX. A message is sent to UNIX as a result of an invocation of the *Send* operation on the *NetIO* object. Also, there is a single thread that is dedicated to distributing incoming messages from the UNIX environment to the designated recipients.

The two environments access a common Ethernet, although any interconnect that supports the TCP/IP protocols could be used. User Datagram Protocol (UDP, [Postel 80]) packets are actually sent over the Ethernet between the two environments. Under UNIX a network server that sends and receives UDP packets implements the AUL protocol. User processes then communicate with this network server by means of standard UNIX IPC.

5.5 Evaluation of Alpha Communication Architecture

The six protocols just described were necessary and sufficient to meet the requirements of Alpha and its first applications. Therefore, they provide a substantial basis on which to evaluate the value and efficacy of the communication architecture chosen for Alpha.

†*Defense Advanced Research Projects Agency.*

5.5.1 Protocol Definition Model

First, the model itself—the CVM, the mechanisms, and protocols expressed as state machines—seems appropriate for present purposes. All of the protocols could be broken down into appropriately small steps (action routines) that respond to specific events. The mechanisms have been adequate, but could stand some improvement. They will be discussed in more detail below.

A more general finite state machine model was originally considered for use in Alpha, but since the simpler model seemed capable of providing the required functions more efficiently, the simpler model was chosen. Specifically, the notion of allowing *epsilon-transitions* in a state transition graph was considered and rejected (for the present). These transitions permit specified state transitions to occur "spontaneously"—that is, the transitions are not triggered by explicit events from the *TokenQueue*.

Epsilon-transitions would allow a significantly different approach to be used in structuring protocols. For instance, under Alpha's current finite state machine model, if a protocol provides a conversation facility for two communicating peers, then the finite state machine's current state uniquely describes *the* current state of the conversation. If non-determinism—in the form of epsilon-transitions—were permitted, then a given conversation could be in any of a number of states at any given time. Non-determinism would allow each state to assume that a specific set of conditions were satisfied when the state was entered. Actions could then be carried out based on those assumptions, and, if later events invalidate any of the assumed conditions, the protocol state diagram can be defined so that there are no valid state transitions from that state for those events.

Successfully applying this approach might reduce the amount of bookkeeping and overhead involved in writing a protocol. However, this approach would also be costly:

1. more state transitions would have to be examined and action routines executed (since more than a single transition would be possible at any time, compared to exactly one transition in the current model);
2. protocol behavior would be much more difficult to analyze, once again, due to the potentially increased number of states that must be considered at any given time for a single protocol conversation; and
3. since each action routine can take actions that are visible (e.g., sending packets or sending commands to the AP), having more than one action routine execute for any given conversation could be very confusing.

While it may be interesting to explore the use of non-determinism in protocol design and implementation, it does not seem to be urgent, or even necessary, at this time in order to meet Alpha's (non-trivial) requirements.

5.5.2 Mechanisms

The functions provided by the mechanisms were generally sufficient, but some improvements were needed. In particular, although the mechanisms allowed packets to be encapsulated by means of adding headers and trailers† to existing packets, there was inadequate provision for stripping this information from received packets. It is clear that there is a hardware advan-

---

†*A header is a group of fields preceding the encapsulated data, while a trailer is a group of fields following the encapsulated data.*

tage that allows the encapsulation to be performed efficiently, and no such advantage exists for stripping the encapsulation. However, in the interest of providing a uniform and complete set of mechanisms that provide well-defined, complementary functions and obey typical rules of abstraction—such as, hiding irrelevant details—the mechanisms should conceal this fact.

Aside from function, the speed of the mechanisms must also be considered. To date, the speed of the mechanisms has been adequate, but it is somewhat slow. The overhead imposed by the CVM model to compose individual mechanisms is minimal, and the mechanisms that manipulate the token queue are efficient. However, the hardware architecture imposes considerable inefficiency in some of the packet handling mechanisms. In particular, the routines that create and receive packets must do excessive copying because the memory used by the Ethernet controller is not directly accessible by the AP or CP. In addition, the mechanisms that manipulate the interval timer are surprisingly slow. They provide the semantics that are desired, but the underlying hardware does not match these semantics.

5.5.3 Flexibility

The protocol definition/CVM model provided excellent flexibility. Simple protocols could be defined and integrated into the CPs suite in a matter of hours or days. And simple protocols were easy to write, which is always a useful property because, to date, simple protocols have outnumbered complex protocols. A protocol definition language might reduce the development time for protocols even further.

5.5.4 Language

All of the protocols described were hand-coded. Nonetheless, they were coded in a stylized manner in order to evaluate the feasibility of developing a protocol definition language. The stylized approach did seem useful; it fit the job nicely. And several “idioms” did develop during the course of the definition of these protocols. These can be hidden/subsumed by a programming language.

Overall, a language seems useful, but it does not look like it will improve performance. It will however simplify the specification of the protocol and therefore reduce the time required to write the definition and also reduce the opportunity for minor bookkeeping errors.

5.5.5 Hardware

The hardware architecture was responsible for several problems in developing the mechanisms and the protocols. First of all, Ethernet is inappropriate as a global bus for Alpha, or any real-time system for that matter, because it cannot provide a deterministic, or at least a known, delivery time for any message. If there is no global clock—as is the case with Alpha, a distributed real-time system needs to know how long messages spend in transit. Currently, average delivery times are measured and then used as if they are deterministic. This is an approximation and will occasionally be wrong.

Other problems result from the Intel 82586 LAN co-processor used on the SMI Ethernet Controller Board. The controller is capable of selectively receiving multicast packets over the Ethernet. However, due to the method used to load and change valid multicast addresses for a node, multicast packets for a node may be frequently missed during changes to the multicast address table. Unfortunately, for the protocols implemented so far—in particular, the RI and TMAR protocols—an objectionable number of packets were missed due to this problem (around ten percent for one simple test program)†. As a result, the hardware multicast filtering capabilities are not used; instead, the filtering is done in software. This, of course, is costly in terms of performance.

Finally, most, if not all, of the protocols are complicated due to the fact that packets can be lost because of the nature of Ethernet (e.g., two simultaneous packet transmissions will result in a collision, which garbles both packets and which may not be detected by both transmitters). Typically, the protocols are more complex because states are added to cope with the possibility of losing a packet, and, as states are added, the number of situations (states) that must be considered for each event that occurs increases. In addition, the additional states and checks that are required often reduce the performance of the protocol.

5.6 Future Directions

The future work in Alpha communications can be broken down into four categories. First, the CVM and underlying mechanisms can be improved. The CVM model can be extended to include non-determinism, but as was pointed out above, this would open up a new research area, rather than make a short-term contribution to the Communication Facility.

Second, the Alpha hardware architecture should be improved as much as possible. Most of the changes that could be made in this area have been referred to earlier. Specifically, as many of the following changes as possible should be made:

- replace Ethernet global bus with a bus that can provide deterministic or known delivery times for packets;
- add dual-ported memory that may be accessed by both the Ethernet controller and the AP and CP processors — this will eliminate a significant amount of processing;
- provide a hardware interval timer — the interval timer chip should manage a queue of timed events (that is, events that are requested to occur at specific times) and these events should be ordered according to their event times; it should be possible to specify new events that are correctly inserted into the queue or to remove previously defined events from the queue; one or more parameters should optionally be associated with each event and should be returned by the interval timer when the timer for an event expires; given the events in the queue, the chip should always determine the parameters needed to respond at the right time for the event at the head of the ordered timed event queue; currently, the mechanisms that implement the interval timer functions are time-consuming and could be significantly improved with better hardware support; this is particularly important since every protocol to date has made extensive use of the interval timer mechanisms;
- employ an improved Ethernet controller — better multicast support will reduce the burden on the software and improve performance;

† *Intel reports another problem with the 82586 LAN co-processor that also results in missed multicast packets Due to this problem, an estimated five percent of the multicast packets addressed to a node may be missed.*

- increase memory bandwidth or buffering on the Ethernet controller — this would reduce or eliminate the underrun and overrun† problems that have been observed in the existing implementation; the underruns and overruns do not "break" a protocol, but they do result in retransmissions that would otherwise be unnecessary; and they occur often enough to warrant attention.

Third, a protocol definition language can be developed. Enough experience has been gathered to permit a language to be defined. This will simplify the task of specifying a protocol and will also reduce the opportunity for error by removing many of the more mechanical, detailed tasks from the protocol designer (this includes the fact that it will insulate the programmer from details concerning the communication mechanisms).

Fourth, much more work remains to be done on protocols. The existing protocols can be refined, and several different trade-offs can be evaluated. For instance, many protocols can consume more bus bandwidth in order to reduce the response time for some action. Assumptions of the reliability of packet delivery can also be explored in order to trade the right amount of error detection for the required performance, remembering that these trade-offs must be made with the priorities that are important for a real-time systems, not a general-purpose computing system.

In addition, a number of new protocols must be developed. Most prominently, the preliminary support for object replication should be improved and support must be developed to support distributed transactions. Part of the transaction support—to handle orphan detection and elimination—will be very similar to the TMAR protocol, and part of it—to handle coordination of committing and aborting transactions—will probably be traditional.

Fifth, steps should be taken to schedule activities in the Communication Co-Processor in the same manner as threads are scheduled in the AP. The architectural changes described earlier should be made to support this end. If the threads on the AP could efficiently access memory belonging to the CP and the Ethernet controller, then they could carry out many of the action routines in response to communication events themselves. Coordination would need to be provided for access to shared data structures and devices, of course, but otherwise, threads could proceed to carry out their own invocations as best suited the application. In general, the object/thread model should be pushed as far as possible into any of the subsystems in order to take greatest advantage of the sophisticated scheduling done by Alpha. (It is impossible to cheaply imitate this scheduling behavior because it is so compute-intensive. In fact, Alpha dedicates a separate processor to perform this function for the AP. It would be quite possible that the CP, or any other subsystem, would spend most of its time scheduling if an attempt was made to replicate the same algorithm on another subsystem.)

Finally, all of the previous changes define areas for the continued research and development of the Alpha Communication Subsystem. However, at some point in the future, Alpha may require more of a production, as opposed to experimental, facility for communication. This will change a number of requirements and priorities in designing the subsystem. Specifically, flex-

†*The LAN co-processor that is used as an Ethernet controller constructs packets based on information in packet descriptors as they are transmitted. If it cannot read data from memory quickly enough to sustain the transmission, then the packet cannot be completed and the transmission is aborted. This is referred to as an underrun. Similarly, when a packet is received, it is placed directly in memory as specified by a set of packet descriptors. If there is insufficient memory bandwidth available to place the data in memory as it arrives, then some data will be lost. This is referred to as an overrun and results in the loss of all of the information in the partially received pocket.*

ibility may not be important in a production version of Alpha. If it imposes a sufficiently large cost, it could be eliminated. The strict interfaces that currently exist might also be broken down to some extent in order to improve performance. Some work must be done in the future to anticipate how to transition Alpha's research results into practice in a production system.

5.7 Summary

Alpha is a distributed operating system and must therefore support communication functions among a number of separate processing nodes. The Alpha programming model defines these communication functions. Since Alpha is a research operating system that, among other things, desires to explore several, possibly diverse, approaches to providing these functions, the Alpha communication architecture emphasized flexibility and adaptability. Six specific protocols have been designed and implemented to provide the necessary communication functions. While most of these protocols have been straightforward, one (TMAR) provided fairly unusual functions in a decentralized manner.

The implementation of the first set of Alpha protocols has demonstrated the viability of the communication architecture and has made Alpha a truly distributed system with unique features. Based on the experience gained from this effort, a number of future directions have been identified—most dealing with improvements or modifications to the architecture. A foundation has been laid to support exploration into a number of communication protocols to support the Alpha object/thread paradigm, in general, and the use of atomic transactions and object replication, in particular.

6 Storage Management Subsystem

The final subsystem in Alpha is concerned with the management of storage resources, and can be broken down into two major parts based on the type of storage that is being managed—i.e., *primary storage* and *secondary storage*.

The first section of this chapter describes the services that Alpha provides for the management of the Application Processor's primary memory. This includes the management of physical memory pages, the management of virtual memory spaces, and the handling of virtual to physical address translations. This chapter's second section describes the Alpha secondary storage facility.

6.1 Primary Storage Management

This describes the manner in which the primary memory of an Application Processor is managed by the Alpha kernel. The management strategies applied to physical and virtual memory are described here, as well as the management of the processor's translation tables and the manner in which virtual memory faults are handled. But first, an overview of the testbed's physical memory structure is given, as well as a definition of the requirements that Alpha imposes on its memory subsystem.

As indicated earlier, the Sun-2 processing element supports a collection of 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 (corresponding to a virtual 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 system software. This management is one of the more difficult aspects of the kernel's virtual memory management function.

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 can be 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 27 illustrates the decomposition of virtual memory spaces in Alpha.

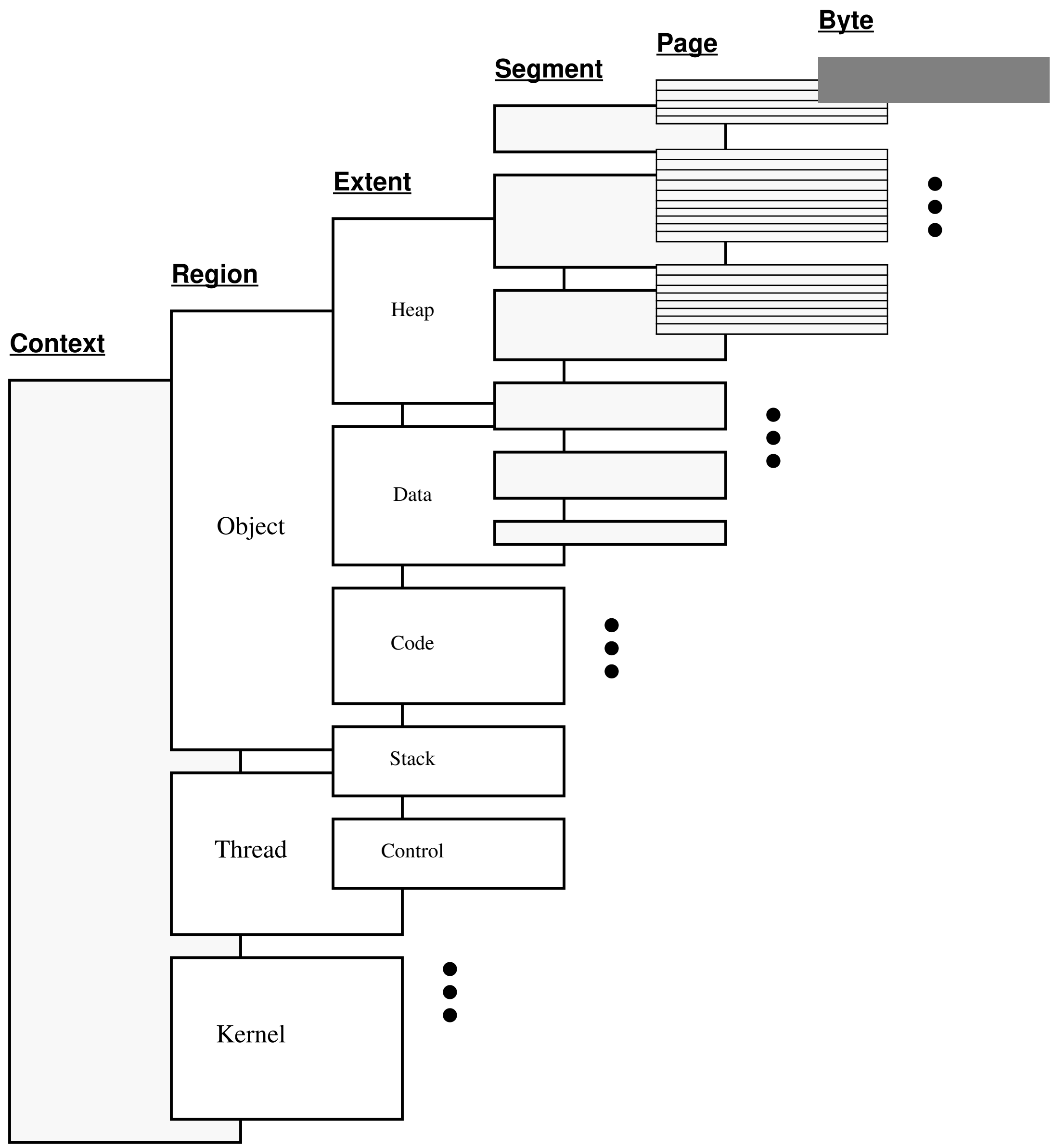


Figure 27: Virtual Memory Hierarchy

The Alpha 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 processing element, the most critically limited resources are physical pages, entries in the SMC, and entries in the PMC. With 1MB of physical memory in the Application Processor, 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 Alpha 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 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.

6.1.1 Physical Memory

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 any one of these separate address spaces. The contexts are initialized by the processing element's hardware and firmware, and then the Alpha kernel takes over and completes the initial set-up of the virtual memory facility.

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 10). The same load module is used for each node in the system; differentiation based on a specific nodes 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 Subsection 3.2.1.1) 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.

6.1.2 Virtual Memory

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 28 illustrates layout of these regions a virtual address space. The following paragraphs provide descriptions of each of these memory regions, their contents and intended use.

6.1.2.1 Kernel Region

A context's kernel memory region consists of two main parts—the *monitor* and the *Kernel Proper*. The monitor portion 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's memory extents are *wired down (*i.e., not available for page replacement), so they are always loaded in the Application Processor's MMU.

The Kernel Proper portion of the kernel memory region 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.

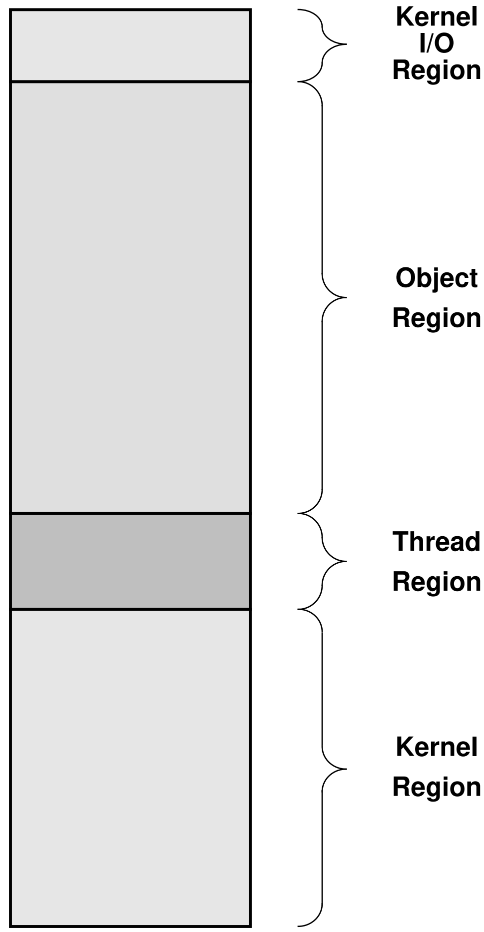


Figure 28: Virtual Address Space Layout

The last two extents of the kernel memory region 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 control blocks 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 control block that represents the kernel (i.e., the KCB) provides the necessary virtual memory structures to permit the kernel page heap extent to be paged.

The kernel memory region occupies the bottom portion of each context, and this memory is protected to allow access to the kernel region pages only in supervisor mode. All client 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 29 illustrates the virtual memory layout of the kernel region.

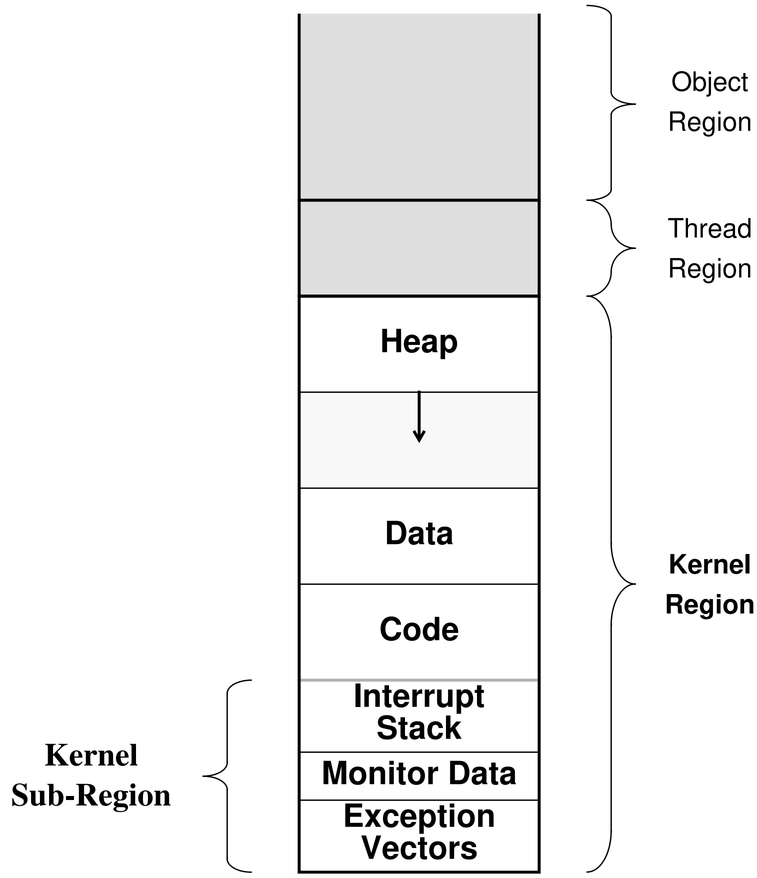


Figure 29: Kernel Virtual Memory Region

6.1.2.2 Client Thread Region

The thread memory 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 thread'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 30 illustrates the layout of the client thread region of a context.

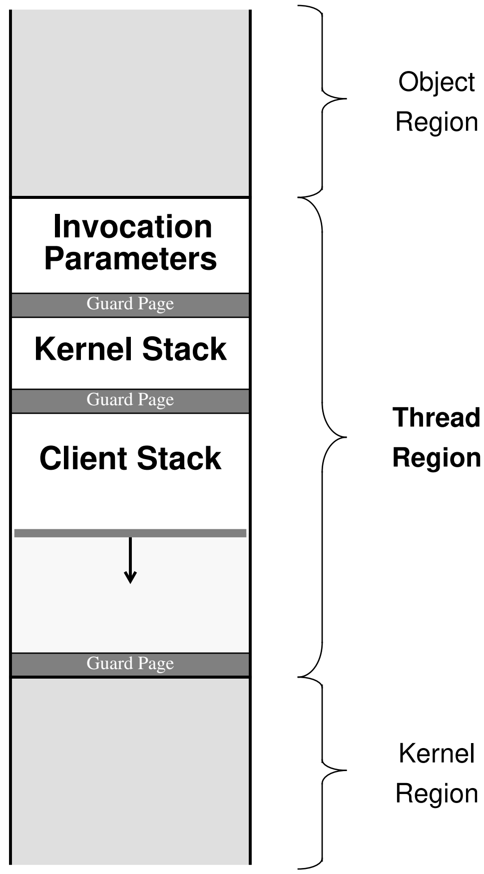


Figure 30: Thread Virtual Memory Region

For a client thread to execute, it must be loaded into a context slot within the MMU. To do this, the kernel must access the thread's control block 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 thread to be activated. Also, just as pages of physical memory can be wired down, so too can certain contexts be made ineligible for removal from the SMC. In this way it is possible to ensure that a thread's address space can be made ready for execution by only changing the context register.

6.1.2.3 Client Object Region

The client object memory region is the place in a context where a thread's currently active object is located. When an operation is invoked by a thread on a local object, 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 located and the thread resumes execution at that point 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.

The client object memory 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 an object's dynamic storage is allocated and this extent is also protected in user read/write mode. Figure 31 provides an illustration of a client thread's region of a context.

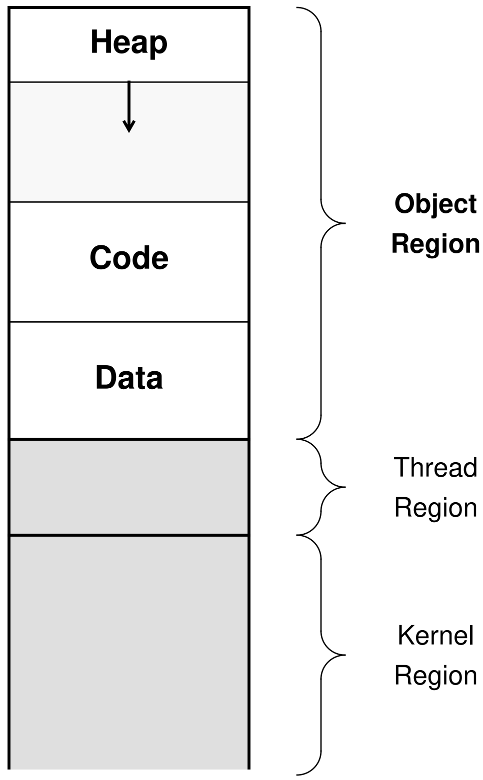


Figure 31: Object Virtual Memory Region

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 control blocks 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 control blocks.

6.1.2.4 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.

6.1.3 Address Translation Table

The Application Processor's SMC and PMC are not large enough to contain all of the segment map and page map entries required for each 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 control blocks, 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 on activation. 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 only allow the modification of translation table mappings for the currently active 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 control blocks, 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 is 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 are 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 control blocks 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.

6.1.4 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 then *unstuck* (i.e., made available for replacement) 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 longer any (virtual or physical) memory available 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.

6.2 Secondary Storage Management

In Alpha, secondary memory is managed by the kernel's Secondary Storage Facility (see Figure 2). 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 maintain, within secondary storage, images of primary memory extents.

Each memory extent in Alpha has a *storage object* associated with it (that is referenced by an extent descriptor data structure). A storage object acts as a repository for one or more memory extents. Each storage object is physically located in either the transient or permanent area of secondary storage provided by the Secondary Storage Facility, and these images may be updated atomically. These features of secondary storage provide support for the attributes associated with the object programming abstraction.

6.2.1 Secondary Storage Facility Interface

The Secondary Storage Facility provides its services to the rest of the kernel by way of a an object-like interface, 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 the facility and its 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 control block. The Secondary Storage Facility handles the operation invocations directed towards the storage 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 control block for this storage object, enters it in the local Dictionary, and returns the Alphabit identifier for the newly created storage object. In Release 1 of Alpha 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 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 control block associated with it in the kernel.

In Alpha, replicas of the Storage Management Facility are implemented on those nodes with Secondary Storage Subsystems (i.e., the nodes that have secondary storage devices and Storage Management Co-Processors). 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 Storage Management Co-Processor. Taken together, these hardware units can be viewed as a disk with a programmable, caching controller—just as the Communication Co-Processor and the Ethernet interface in the Communication Subsystem can be considered an intelligent network interface. Figure 32 provides an indication of the relationship of the secondary storage devices to the testbed system.

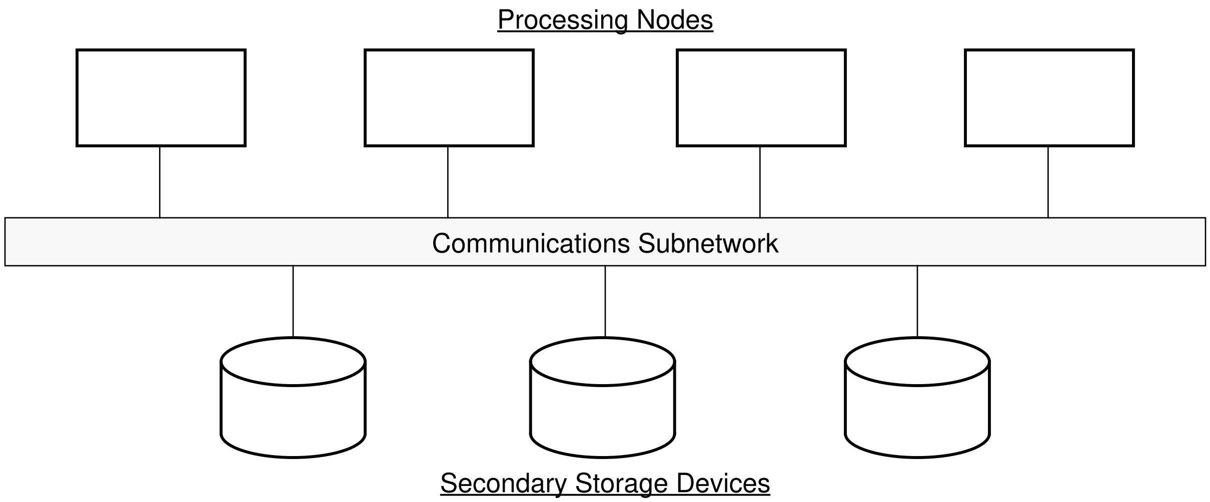


Figure 32: Logical View of Secondary Storage in Alpha

The Secondary Storage Subsystem 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 Subsystem is responsible for performing all of the caching of control and data information associated with storage objects. It is responsible for the "short-circuiting" of 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 when the page is read/write protected but has not been modified since the last time it was read from its storage object. Additionally, the concurrency provided by the Storage Management Co-Processor makes possible a number of optimizations. For example, the Secondary 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.

6.2.2 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, control blocks 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 control block 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 3.2.1.1) 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.

6.2.3 Low-Level Storage Management Details

While the Secondary Storage Facility was designed to support a wide range of secondary storage technologies, the only one addressed in Release 1 of Alpha is standard rotating magnetic media devices—i.e., Winchester disks. This subsection describes the manner in which information is represented on the physical device(s) in Release 1 of Alpha. Since an Alpha system is expected to be reliable, the issue of representation is particularly important. The chosen format represents a convention, which all software and hardware components must adhere to.

The major fact which gives the disk format conventions importance, is the fact that all the components of a system must occasionally be initialized and go through a start-up phase. This is as true of the data on the secondary storage devices as it is of any other component. However, the data are distinguished by the times at which this occurs. A data collection is initialized at the discretion of a system administrator, and the initialization is performed by a running system. A system is initialized every time that it is started up, and every time that circumstances force a restart, but the data in secondary storage are not initialized at that time. Instead, the data are treated as a special resource whose logical correctness must be assured (or assurable to some degree of confidence).

The considerations that went into the design of the low-level storage formats are (in order of importance): robustness, timeliness, restartability, speed efficiency, and space efficiency. The robustness item reflects the fact that hardware failure is of primary concern of Alpha. It is unacceptable that such failures have a high probability of causing undetectable or unrecoverable damage to data. Since some data are of transitory importance, and since some robustness techniques exact a higher performance penalty than others, it was clear that no single technique offered the most appropriate trade-off for all uses. Therefore, Alpha should allow for several different robustness tradeoffs. In this context, timeliness means that the secondary storage subsystem manages its resources based on the time constraints of the threads on whose behalf operations are being performed. Furthermore, some attempt has been made to limit the variance of delay associated with the execution of secondary storage operations. In particular, an attempt was made to help avoid "recursive disasters", where a whole chain of events may (with low probability) all have to be serviced before some single request is satisfied. In Alpha, a Secondary Storage Subsystem should not spend an inordinate amount of time performing the various housekeeping activities in order to resolve the consequences of the last shutdown/failure. Instead, these actions are discretionary, or are performed in parallel with other work (potentially as a lower priority activity which can be interrupted/resumed (or aborted/restarted) consistently with the system's higher-level goals).

For Alpha Release 1, it was decided to implement a Secondary Storage Subsystem which was extensible to high sophistication, and to very large disks, but which could be useful to the project at several levels of partial implementation.

The primary robustness technique of the initial implementation is "stable storage" on a non-replicated disk. This technique involves writing two copies of a given datum to the same disk, each copy separately check-summed and time-stamped. Logging techniques are planned for integration with transaction support, but this was not deemed as appropriate for initial low-level requirements. A disk failure model was adopted, which permits a write of one sector to be corrupted, and a single write of N contiguous sectors may leave corrupted data in two successive sectors. The model assumes these effects occur because the write was in progress at the time of the hardware failure. The model assumes that a failure at any other time will leave the stored data uncorrupted. It is also assumed that the failure will leave the subsystem unable to perform further writes. Given this failure model, it was decided that stable storage of a small datum would involve a single write of three contiguous sectors, with null data stored between two identical copies of the datum. All accesses to such a datum must read both copies, and verify both check-sums, and compare the data. If only one datum is acceptable, the other is "healed." If both data are acceptable, but they differ, then the younger or first is assumed to be dominant, and the other is immediately corrected on disk. It was furthermore decided that stable storage of multi-sector data, that is written one sector at a time, would involve two writes. No guard region is used in these cases. Stable storage of multi-sector data, that is written in a single write, has not been needed for low-level purposes.

A basic consequence of the restartability requirement, and of the failure model, is that all low-level data structures on disk must be weakly self-consistent at all times. By this, it is meant that a restart must not have to perform a complete scan of the low-level data, before knowing for certain that given datum can be used in the normal way. (For example, a directory lookup can be performed, without recovery analysis of any components not encountered by a normal look-up.) This is referred to as "weak" self-consistency, because we are willing to allow storage losses. It is an acceptable consequence of a crash that some bounded amount of empty disk space become inaccessible. It is intended that the processing to detect and correct this loss will be discretionary, or low-priority.

Since the Secondary Storage Subsystem is primarily oriented towards objects and paging, rather than towards small text files, it was decided that allocation would be based on extents. That is, device space would not be treated as equal-sized chunks, but would be treated as varying-length items, potentially as large as an entire disk. This appeared to satisfy several criteria. By allowing compact descriptions of larger entities, access structures could be smaller and therefore traversed in fewer steps, or even completely cached. As will be seen, access times could be bounded, and large data transfers could be performed with considerable directness, a performance consideration, since cached writes have bad failure properties. The structure of objects in memory can (usually) be directly mapped onto disk extents. Since lists of free space are compact, in-memory representations could be kept in address order, facilitating contiguous allocation. A high-efficiency data base subsystem could be allowed low-level control within a dedicated extent.

It was also decided that version codes would be placed into data structures, in such a way that interoperability problems would not be caused by removable disk packs.

6.2.3.1 Device Header

It is important that every writable device be identified as to its history. Therefore, Alpha writes a label at the head of each device (disk) when the device is formatted. This label uses the stable storage convention, and it is intended that it contain a variety of information, such as:

- the version of the conventions holding for the contained data
- the time-stamp of the formatting operation
- a time-stamp bounding the last time that the device was written to
- identification of the disk, and some history, if removable
- information bounding the range(s) of capabilities present on the disk

Information concerning the faults on the device has been stored separately, in the manner which is standard for the device in the Alpha testbed. This was intended to allow the use of a commercially obtained diagnostic program, which deals with "bad blocks" in an acceptable fashion.

Information concerning the geometry of the device is already present in the fault map, and has not yet been represented in the Alpha label.

6.2.3.2 Space Allocation

Since space is dealt with in extents, an empty disk contains exactly one extent. (Actually, it contains exactly two, because some structures are kept in the middle of the disk, to minimize the access time of seeks to these structures.) Therefore, the initial "free list" is only a few bytes long (16 bytes, in our case). When an extent is allocated to some purpose, the initial free list neither grows nor shrinks; instead, it describes less free space, since one of the length counts in the list is decreased.

Matters become more complicated when an extent is freed. The simple solution would be to merge the newly freed extent into one of the existing free extents. However, due to fragmentation, the newly freed extent may not be contiguous to an existing extent, and the free list must become longer. It is in this manner that the list lengthens, and in theory it could lengthen until we are spending 8 bytes to describe every second disk sector: that is, one part in 128 of the disk. In practice, no such explosion is likely, given the expected pattern of demand from dedicated real-time applications, and the time scales involved in that many demands for space. The initial implementation allows the free list to grow to 62 extents, the maximum that will fit into a single disk sector. The data structure is designed to be enlarged, and expansion space has been left on the disk. A background or privileged "scavenger" program is envisaged, which is capable of moving small disk objects around on the disk, thus allowing free extents to be abutted and merged. As long as an implementation allows much more free-list space than the "equilibrium" amount, this approach should have no more problems than any alternative. This has been weighed against the benefits, which include highly contiguous allocation (for fast reads and writes), and fast allocation (since alternative approaches must allocate a larger number of smaller pieces).

Given that there is such a free-list, it was decided to keep it sorted in address order (for easy merging on deallocations), rather than in size order (for best-fit allocation).

Next, it was determined that deallocation could be *lazy*—i.e., the free-list can become disordered. A deallocation merely places the freed extent(s) at the rear of the free-list, and increment a "dirty count" which is internal to the free-list. When the request interpreter has no outstanding work, it will check if the free-list needs to be rationalized, and perform this operation. Rationalizing removes any zero-length entries, and performs the equivalent of a sort. It would also be quite easy to allow rationalization to be interrupted when work comes in.

Ideally, the in-memory free-list would correspond directly to the on-disk free-list. However, it is acceptable to use lazy writes after deallocations, since a crash will at worst lose the deallocated space. The space is not lost forever, of course: it is only lost until a scavenger process computes a correct free-list from the access structures, and writes the correct free-list over the incorrect one. This scavenger does not have to be run immediately after a restart, in accordance with the requirement for restartability.

Lazy writes are also acceptable after free-list rationalization. However, if space was allocated rather than deallocated, lazy writes could cause a problem. After a crash, the free-list on disk might claim the space was free, while the access structures claim that some viable object resides there. This is acceptable if we are willing to run a scavenge procedure on restart, but as has already been stated, we do not want to do this. An alternative is to always write allocation information immediately out to disk, either by writing the free-list, or by using a logging technique. This is speed-limiting, so we prefer to avoid it. The alternative which has been chosen is to write the free-list to disk, but only if an optimization method fails.

The optimization method is simply a pre-allocation scheme. Some number of disk blocks are allocated on start-up, and the extent(s) so allocated are noted in memory. Any sufficiently small allocation requests may be given the pre-allocated space, without causing a disk write, and the pre-allocation state is then marked for lazy renewal. As before, a crash may temporarily lose some amount of disk space. The only time penalty is that start-up/restart must now perform allocations, thus slowing restart by one disk write. When an object is created, the allocation process will involve exactly one disk write, which typically will be performed lazily, that is, at some time after the allocation has been reported as successful.

The free-list data structure is kept in the middle of the disk, to minimize seek times. The middle of the disk can be found because the software knows the disk geometry, either from the disk label, or by other means.

6.2.3.3 Capability Mapping

The "name" of an object on secondary storage, is the Alpha capability which was specified when the object was created. At the level which the kernel deals with, no other style of name is interesting, and it is assumed that many objects would never have any other sort of name. Of course, higher levels of software are free to maintain mapping data, which allows application-specific names (or human-readable names) to be rendered into capabilities. In general, enough tools will exist, and an application could, for example, maintain a tree-structure in the style of UNIX. No specific support for this needs to exist at the kernel level.

The Secondary Storage Subsystem must nevertheless maintain one mapping, which is the mapping from capabilities to objects. This capability mapping information—the access structure—must be kept reliably on the disk. Some systems have used B-trees to do this, however, in Alpha this approach was rejected on the grounds of unpredictability: an update may cause an update, and this may ripple through some large fraction of the B-tree. B-trees do have the advantage of allowing sequential access, but this is not needed: the indexes are capabilities. B-trees are also bulky, in the sense that root nodes must store at least portions of the indexes, whereas the scheme below does not. Also, B-trees are somewhat difficult in an environment where one thread may be updating the tree while others are using it. (The difficulty is another aspect of the ripple problem: deadlock can ensue if a writer does not initially lock almost the whole tree.)

It was decided that we would use a two-level version of a technique called *extendible hashing*. This technique is applicable when the concepts of "next" and "previous" index are not needed. This is because the method uses hashing, which destroys sequential properties, but has the advantage that the index values (here, capabilities) are more uniformly distributed across the access structure. This seemed useful because capabilities tend to have internal structure generated by upcounters, and this means that a capability treated as a bit pattern may have difficult statistical properties. By hashing these bit patterns, a useful statistical property is at hand.

The basic method requires that a *root map* be kept, which contains N *leaf pointers*. N must be a power of two, for reasons which will be apparent shortly. A lookup is done by indexing (with bits from a hashed capability) to select one of the leaf pointers. A leaf pointer is merely enough information to allow us to compute the disk address of any of `L` *leaves,* where a leaf is a data structure occupying `S` disk sectors. (In our initial implementation, `S` is 1.) Each leaf contains a hash-searchable list of capabilities, and attached to each capability is the disk address of an *object descriptor*. (The object descriptor is stored contiguously with (one of) the extent(s) which make up the actual object.)

In the initial implementation, this structure is capable of addressing 128 * 63 = 8064 object descriptors. This was acceptable as a first cut, since the structure would not saturate before the 64 megabyte disk was full, unless the average object took up less than 7800 bytes. It is not acceptable for larger disks, but there are good techniques for expanding this basic structure.

Several techniques fall under the heading of "efficiency" methods. For example, the leaf pointers could be compressed to `B` bits, by assuming that all leaves are stored in some bounded region of the disk. (That is, a base pointer having more than `B` bits is added to the leaf pointer, to convert it to a disk address.) Also, the leaves could have capabilities in sorted order, which allows compression techniques, at some penalty in (in-memory) search time. Collectively, efficiency techniques can expand the structure by a factor of (say) four, which is approximately ten times short of the desired size. In general, these methods seem most appropriate after an acceptable structure has been implemented, and time is found to make small improvements in efficiency.

The crude approach is to use a larger fixed amount of disk space. Direct scaling of each leaf, and of the root map, to four kilobytes each, would be enough. This is actually inefficient, because the structure is likely to be very sparsely populated, leading to inefficient use of disk bandwidth and inefficient caches.

The reasonable thing is to use a compact layout, and expand dynamically to use more disk space as needed. Extendible hashing offers a pleasant algorithm for this, which avoids the potentially high costs of dynamic extension. The article suggests that the root map be initially quite small, and contain identical leaf pointers. (For example, it might be of size 4, with all four pointers pointing to the same leaf.) As objects are created, leaves become more full, and eventually a leaf will overflow. At this time, we determine `P`, the number of pointers to this leaf. If `P` is greater than 1, then it will be a power of two. In this case, we split the leaf into two leaves, and adjust half of the `P` pointers to point to one of the new leaves, and adjust the other half of the pointers so that they point to the other of the new leaves. (The entries in the original leaf will be assigned one way or the other according to where their hash now falls.)

If `P` was one, then only one slot in the root map yielded that leaf. This case is handled by creating a double sized root map, and copying each pointer in the old map to two adjacent positions in the new map. The `P` value is now two, and the overflow algorithm given above can be applied. It can now be seen why the root map was required to have a size which is a power of two. The lookup algorithm takes the hash of a capability, and uses several bits of the hash value as an index. When the root map grows (or shrinks), the lookup simply has to start taking one more (or less) bit from the hash value. No other change needs to be made to insure that previously stored capabilities can still be found.

This scheme is logically simple, and should be quite fast, because a leaf split involves at most three disk writes (two leaves and the new root). It also has highly localized disk activity, not only during leaf splits, but at all times, including restart. The root map stays small enough that it can be kept in memory: hence, a lookup performs at most one disk read to obtain the address of an object descriptor. The phrase “at most” is used because caching techniques will be applied to leaves. It should be noted that extendible hashing allows lazy load balancing, so that all leaves can be kept at a “reasonable” fullness. In a system which is operating below overload, it should be possible to assure that all disk splits are performed lazily, as part of the background load balancing. This would mean that object creation could never cause more than one write to the access structure—surely, some sort of minimum, since the object’s existence is not committed until some write occurs.

It is intended that leaves will be kept in the middle of the disk, in a statically-allocated region of adequate size. We are not aware of any convincing argument about how to guarantee an advantage from allowing leaves to be scattered. (There are non-extent-based systems, which keep a local map on each cylinder of their disk, but this does not seem appropriate for Alpha.)

It was stated earlier that the object descriptors are not stored in the leaves, but are instead stored at the front of each object (or front portion thereof). Other systems have decided otherwise, and argued that this allowed a lookup to take one less disk read. It has also been argued that in-leaf storage is very efficient for the most common operation on these file systems, namely, a directory scan. However, the Alpha system will not attempt such scans. Also, if the object data are stored in the leaves, then the access structure becomes bulkier.

In fact, an Alpha object descriptor is a fair-sized varying-length structure, since Release 1 objects may consist of up to four segments, each of which may be stored on disk as multiple disk extents. This size, and the algorithmic implications of varying size, persuaded us that object descriptors should be stored separately. This was expected to be useful as the implementation of various higher-level schemes eventually required the augmentation of the object descriptor format.

It was also expected that later implementations would be on machines with enough memory to allow whole-track transfers, and track caching. (This is done for the speed advantage, since a track operation does not necessarily take any longer than a sector operation.) On such a system, the design is expected to show performance improvements that might not be realized by alternate designs.

7 Acknowledgments

Many people contributed to this effort, and many others continue to contribute as the Archons project enters its next phase.

Doug Jensen is the founder of the Archons project. He obtained the support for the project over its 10-year existence, and provided most of the fundamental philosophy and concepts for Alpha. Doug continues to provide support and guidance for this research effort as Release 2 of the Alpha operating system is being developed at Kendall Square Research Corporation.

In his brief visit with the Archons project, Martin McKendry initiated the implementation effort that has become Alpha and provided many of the initial implementation concepts.

Donald Lindsay worked on the secondary storage subsystem and contributed text to Subsection 6.2.3 of this report.

Sam Shipman worked on various subsystems and support tools, and contributed to refining and completing the system design and implementation. David Maynard also worked on various aspects of the system's design and implementation, and acted as the project liaison with General Dynamics during our joint C^2 demonstration effort. David is now working on a thesis in the area of real-time scheduling for decentralized computers with multiprocessor nodes. Huay-Yong Wang worked on the scheduling subsystem and the design and implementation of various other system functions.

Other project members that contributed to this effort are Jeff Trull, Chuck Kollar, Bruce Taylor, and Dan Reiner. Thanks are due to Tom Lawrence and Dick Metzger, the Archons project's prime sponsors at the Rome Air Development Center.

References

[Accetta 86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix.* July, 1986.

[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.

[Bochmann 78] Bochmann, G. V.
Finite State Description of Communication Protocols.
In *Proceedings of the Computer Network Protocols Symposium*, pages F3-1-F3-11. February, 1978.

[Bochmann 80a] Bochmann, G. V.
A General Transition Model for Protocols and Communication Services.
IEEE Transactions on Communications COM-28(4):643-650, April, 1980.

[Bochmann 80b] Bochmann, G. V. and Merlin, P.
On the Construction of Communication Protocols.
In *Proceedings of the Fifth International Conference on Computer Communication*, pages 371-378. October, 1980.

[Bochmann 82] Bochmann, G. V., et. al.
Experience with Formal Specifications Using an Extended State Transition Model.
IEEE Transactions on Communications COM-30(12):2506-2513, December, 1982.

[Brand 83] Brand, D., and Zafiropulo, P.
On Communicating Finite State Machines.
Journal of the ACM 30(2):323-342, April, 1983.

[Chesson 80] Chesson, G. L. and Fraser, A. G.
Datakit Network Architecture.
In *COMPCON 80 — Spring, IEEE Computer Society International Conference*, pages 59-61. February, 1980.

[Chesson 87] Chesson, G. L.
The Protocol Engine Project.
UNIX Review 5(9):70-77, September, 1987.

[CCITT 85] CCITT.
Recommendations Z.100 to Z.104: Specification and Description Language.
In *Red Book.* Geneva, Switzerland: ITU, 1985.

[Clark 82] Clark, D. D.
Modularity and Efficiency in Protocol Implementation.
RFC 817, MIT Laboratory for Computer Science, July, 1982.

[Clark 88] Clark, R. K., Kegley, R. B., Keleher, P. J., Maynard, D. P., Northcutt, J. D., Shipman, S. E. and Zimmerman, B. A.
An Example Real-Time Command and Control Application.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March 1988.

[CMC 85] Communication Machinery Corporation.
Ethernet Node Processor ENP-30 User's Guide.
Communication Machinery Corporation, April, 1985.

[Danthine 81] Danthine, A.
Design Principles of Communication Protocols.
In *Data Communications and Computer Networks*, pages 257-273, IFIP, 1981.

[Fleisch 81] Fleisch, B. D.
An Architecture for Pup Services on a Distributed Operating System.
Operating Systems Review 15(1):26-44, January, 1981.

[Fletcher 78] Fletcher, J. G. and Watson, R. W.
Mechanisms for a Reliable Timer-Based Protocol.
In *Proceedings of the Computer Network Protocols Symposium*, pages C5-1-C5-17. 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.

[Herlihy 84] Herlihy, M. P.
Replication Methods for Abstract Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[Hopcroft 79] Hopcroft, J. E. and Ullman, J. D.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley Publishing Company, 1979.

[Gouda 85] Gouda, M. G.
Protocol Validation by Fair Progress State Exploration.
Computer Networks and ISDN Systems 9(5):353-361, May, 1985.

[ISO 86] International Standards Organization.
Estelle — A Formal Description Technique Based on an Extended State Transition Model.
ISO TC 7/SC 21/WG 1 — FDT, Subgroup B, September 1986.

[Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.

Technical Report, Honeywell System and Research Center, June 1975.

[Jensen 78] Jensen, E. D.
The Honeywell Experimental Distributed Processor — An Overview.
Computer 11(1):28-37, January, 1978.

[Jones 79] Jones, A., Chansler, R., Durham, I., Schwans, K. and Vegdahl, S.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating System Principles*, pages 117-127. ACM, December 1979.

[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.

[Liskov 84] Liskov, B. H.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February, 1984.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May 1986.

[McKendry 84] McKendry, M. S.
The Clouds Project: Reliable Operating Systems for Multicomputers.
Project Report, Georgia Institute of Technology, 1984.

[Nash 83] Nash, S. C.
Automated Implementation of SNA Communication Protocols.
In *Proceedings of IEEE International Conference on Communications*, pages 1316-1322. June, 1983.

[Nelson 81] Nelson, B. J.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, May, 1981.

[Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.

Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.

[Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Northcutt 88d] Northcutt, J. D.
The Alpha Operating System: Kernel Programmer's Interface Manual.
Archons Project Technical Report #88061, Department of Computer Science, Carnegie-Mellon University, June 1988.

[Postel 80] Postel, J. B.
User Datagram Protocol.
RFC 768, USC/Information Sciences Institute, August, 1980.

[Postel 81a] Postel, J. B.
Internet Protocol.
RFC 791, USC/Information Sciences Institute, September, 1981.

[Postel 81b] Postel, J. B.
Transmission Control Protocol.
RFC 793, USC/Information Sciences Institute, September, 1981.

[Ritchie 74] Ritchie, D. M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[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.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April 1988.

[Shrivastava 82] Shrivastava, S. K. and Panzieri, F.
The Design of a Reliable Remote Procedure Call Mechanism.
IEEE Transactions on Computers C-31(7):692-697, July, 1982.

[Sproull 78] Sproull, R. F. and Cohen, D.
High-Level Protocols.
Proceedings of the IEEE 66(11):1371-1386, November, 1978.

[Thornton 64] Thornton, J. E.
Parallel Operation in the Control Data 6600.
In *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 26, part 2, pages 33-40. 1964.

[Wulf 81] Wulf, W. A., Levin, R. and Harbison, S. P.
Hydra/C.mmp: An Experimental Computer System.
McGraw/Hill, New York, 1981.

[Xerox 81] Xerox Corporation
Courier: The Remote Procedure Call Protocol.
Technical Report XSIS 038112, Xerox Corporation, December, 1981.