MK++ Kernel
High Level Design

OSF

January 12, 1996

Revision History:

Revision 1	April 13, 1994	Initial evaluation copy
Revision 2	July 6, 1995	Consistency with first release
Revision 2.1	November 17, 1995	Draft formal evaluation copy
Revision 2.2	January 12, 1996	Formal evaluation copy

This work was sponsored in part by the Advanced Research Projects Agency and the Air Force Materiel Command.

Preface

This document presents the design of the MK++ kernel. Although an effective replacement for the previous Mach kernel, the MK++ kernel represents a complete redesign of Mach in accordance with the assurance requirements for evaluation at the B3 level. For this evaluation, the assurance requirements specify "significant use of layering, abstraction and data hiding."

Accordingly, the design of the MK++ kernel is strictly layered; this means that no layers in the kernel's structure depend upon functionality implemented in a higher layer in the structure. The kernel's design also has an object orientation with the goals of abstraction and data hiding. This document is structured so as to emphasize this layered and object oriented structure.

The kernel is subdivided into layers. Each layer is composed typically of one, but sometimes a small number of, subsystems. A subsystem consists of a related set of objects and mechanisms. Each subsystem is described by its own chapter; the grouping of subsystems into layers is shown in the first chapter ("High Level Design"). Since most layers are composed of a single subsystem, the top to bottom ordering of subsystem chapters can generally be read as a top to bottom decomposition of the kernel.

The first two chapters, *High Level Design* and *System Decomposition*, present overview material. Chapter 1 provides an overview of the kernel's functionality, describes the kernel's subsystems, presents the kernel's layering hierarchy, and describes issues involved in concurrency. This chapter also describes the kernel's general security policy and describes its initialization sequence.

Chapter 2 describes the decomposition of the user-visible kernel abstractions into their constituent pieces. This chapter also describes the mechanisms by which the kernel is entered.

Although the kernel's design is object oriented, the choice of the neutral term "Functional" for the decomposition relates to the following aspects of this decomposition:

- Any given subsystem consists of more than one class of objects, which relate to each other in any number of ways, often having to do with some common abstract functionality they encapsulate.
- Although a class is considered to be defined in a particular subsystem, it is often the case that one class will derive from a class in a lower (but never higher) layer, so objects as a whole (their members and their base class members) are not strictly bound within a single layer.
- In some cases not all of the methods for a given class can be implemented within the layer in which that class lies. In these cases, methods may be implemented in higher layers so as to be able to access objects accessible in those higher layers. Any given method is always considered to be implemented in a particular layer.
- To prove the assertion of strictness of layering, it is necessary to examine which methods (class functions) in what layers may be invoked by any given method of any given object.

The rest of the book's chapters, each of which focuses on a single kernel subsystem, have the following essential elements:

- An overview of the purpose of the component, its functionality and abstractions and its reason for being in its place in the design.
- A statement of the interfaces to objects within the component that are exposed to components outside the component.
- Statements about security relevant aspects of the component with respect to the computer security evaluation of the larger trusted system.
- A description of the component's decomposition and the relationships between those subcomponents.
- A description of concurrency issues the component faces.

Document Typeface Conventions

This document uses the following typographic conventions:

- The following typeface is used to specify object classes: *anonymous_virtual_extent*.
- The following typeface is used to specify object instances: anonymous_virtual_extent.
- The following typeface is used to specify class methods and functions: **anonymous_virtual_extent**.
- The following typeface is used to specify variables: *share*.

For example, the following specifies a class method that requires an object instance and a variable in its argument list:

> *anonymous_virtual_extent*::**anonymous_virtual_extent**
> (anonymous_virtual_extent, *share*).

Note that the document also uses the *italicized* typeface when introducing new terms and concepts.

Contents

Tables and Figures

CHAPTER 2 System Decomposition

The MK++ kernel provides a large set of user-visible entities, including tasks, threads, memory objects, and devices. This chapter describes the decomposition of these entities in terms of the instances of classes exported by specific kernel subsystems. The purpose of this chapter is to provide a context that will aid the reader in understanding the subsequent chapters of this book, each of which focuses on the decomposition of an individual kernel subsystem.

Whereas the following chapters describe the kernel's composition "horizontally," subsystem by subsystem, this chapter provides a "vertical" look at the kernel to illustrate how the various subsystems participate in the composition of the user-visible abstractions. The descriptions presented here are intended to broaden the reader's conceptual understanding of the MK++ kernel and to that end, detail that would interfere with that goal is excluded. Consequently, the presentation contained herein should in no way be considered a complete description or a formal description of the kernel's decomposition; it is merely a summary of the significant components that make up the principal abstractions the kernel exports to the user.

General Characteristics

In this chapter the abstractions that MK++ exports to the user are referred to as *composites* —that is, abstractions composed of lower-level abstractions. Each component abstraction encapsulates specific state with respect to the composite and plays a distinct role in the semantics that govern the composite's functionality and behavior.

Consider the user-visible thread abstraction. A MK++ thread is first and foremost a point of control that executes in the context of an address space. As an executable entity, a thread is subject to the system's scheduling algorithm, hardware interrupts, and the thread's use of the system's event-wait mechanism. All state associated with this aspect of the thread is encapsulated in the thread's *execution point* abstraction, an abstraction exported and managed by the kernel's Processor Scheduling subsystem.

A thread is also a first class kernel object that user-level code can invoke operations on by sending messages to the port that represents the thread. The state associated with this aspect of the thread composite represents the thread as a communication endpoint and is distinct from the thread's execution point state; it is encapsulated in the composite's *kernel object* abstraction which is exported and managed by the Connection Management subsystem.

Connections

An important MK++ concept is that of the *connection*. There are two types of connections: *user ports* represent connections between user tasks, and *kernel objects* represent connections between a user task and the kernel. (The *MK++ Kernel Executive Summary* refers to *kernel ports*; the kernel object contains the state a kernel port would have encapsulated, if there were such entities as kernel ports within the kernel.)

With respect to encapsulated state, the most important difference between the connection types is that a user port contains a message queue for messages pending delivery. Kernel objects do not have message queues. When the user invokes a send operation on a kernel object, the kernel receives the message via that object and immediately invokes the appropriate kernel operation. If the kernel object happens to be the target of a send operation when another send operation is invoked on it, the second send operation finds that the kernel object is locked and blocks until the earlier operation completes and unlocks the kernel object.

There is more to say about the decomposition of user port and kernel object connections, but that discussion will be deferred for the moment to pay attention to the connection locking mechanism.

Synchronizing Access to Connections

Regardless of a given connection's type, a user-level thread may access the connection only if the thread's task holds a right that names the connection. IPC semantics allow multiple tasks to hold send rights to a connection and the kernel protects a connection's state by requiring each send operation to lock the target connection before the operation may manipulate the connection's state. The connection-locking mechanism essentially involves a mutex; if a connection is locked when a given thread invokes a send operation on the connection, that operation blocks in the kernel until the connection is unlocked.

A connection's receiver must also use the connection-locking mechanism when modifying the connection's state. A receiver must, for example, lock the underlying connection when receiving a pending message from the connection or when moving the connection into or out of a port set. A receiver must also lock the connection before destroying it.

The task that holds a connection's receive right can destroy the connection at any time. Though it would be possible to search for and invalidate all associated send rights when destroying a connection, IPC semantics do not dictate this behavior and so MK++ optimizes connection destruction by ignoring a target connection's outstanding send rights. Because the kernel allows a connection's receiver to arbitrarily destroy a connection while other tasks hold send rights to the connection, the kernel must provide a mechanism that allows user threads to attempt to lock connections that may no longer exist.

Figure 3 shows how this behavior is achieved. As shown in the figure, a task's name for a connection, represented by a *user port name space entry*, points to a *capability* which represents the kernel's internal global name for the connection. The capability contains a reference to a *connection lock*. (In the current implementation, connection locks are known as *port locks*.) The figure also shows which subsystems implement and manage which abstractions.

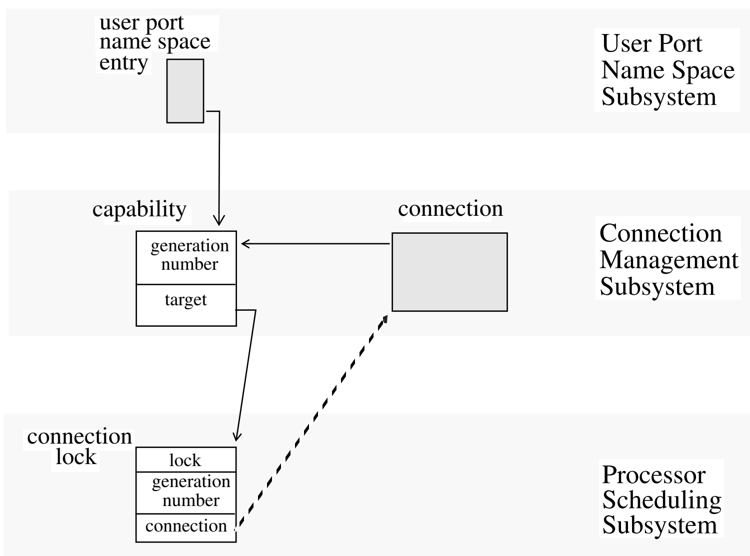


Figure 3 **Connection Lock Mechanism Architecture**

The connection lock contains a pointer to the *connection* it protects. In the figure, this pointer is represented by a dashed arrow to indicate that although the Processor Scheduling subsystem holds this pointer, that subsystem never dereferences the pointer; the pointer is merely returned to the Connection Management subsystem (which supplied the value initially) when the connection lock is locked.

Note that the capability and the connection lock both contain generation numbers. A capability's generation number and target pointer never change, but a connection lock's generation number and connection pointer can be changed if the connection it points to is deallocated. The kernel recycles connection locks; when a connection lock's current connection is deallocated, the kernel is free to assign the connection lock to a new connection. The kernel updates a connection lock's generation number each time the lock is associated with a new connection.

The connection itself contains a copy of a capability that names it. This provides the pointer by which the kernel can manipulate the lock when it has a connection in hand.

With these objects in place, the connection locking mechanism performs the following actions (atomically):

1. The kernel uses the capability to locate the appropriate connection lock.
2. The kernel determines whether or not the capability's generation number matches that of the connection lock.
3. If the numbers match, the kernel locks the connection lock and returns a pointer to the connection. If the numbers do not match, the capability is no longer valid and the kernel returns an error.

Kernel Object Composition

The majority of MK++ user-visible abstractions are first class kernel objects such as tasks, threads, ledgers, memory objects, and devices. User-level threads invoke operations on these objects by sending messages to the connections that are encapsulated within these objects.

In the abstract, a given kernel object includes the following components:

1. A handle that represents the object as a specific and identifiable *managed resource*; this component represents the object as a first class kernel object.
2. One or more components that implement the object's core functionality.
3. The *connection* and *kernel object* components represent the object as a communication endpoint. (The connection component represents the object as a lockable target of IPC message send operations, the kernel object component specifies that the object is a communication endpoint within the kernel—in other words, messages sent to this communication endpoint invoke kernel operations.

Figure 4 shows a *very* simplified version of a thread object's decomposition (a more complete decomposition will be provided later). The components shown in the figure map to the component types listed above as might be expected: the *thread* component represents the thread as a first class kernel object; the *execution point* component encapsulates the thread's core functionality as a point of control whose execution is controlled by the Process Scheduling subsystem.

Kernel Object Invocation

How the kernel's subsystems interact to implement system calls that target first class kernel objects can now be discussed. In most cases, MK++ system calls are implemented as RPCs wherein the user sends a message to the kernel to invoke the operation and receives a reply that indicates the success or failure of the operation. Kernel object invocation can be divided into three phases:

1. Find and lock the object via the user's port name for the object.
2. Invoke the requested operation on the object.
3. Create and send the kernel's reply to the user.

In the following subsections the focus will be on the details surrounding the first two of these phases. The phases are described as they would occur if a user invoked the **thread_get_state** system call on a thread like the one depicted in Figure 4. (The

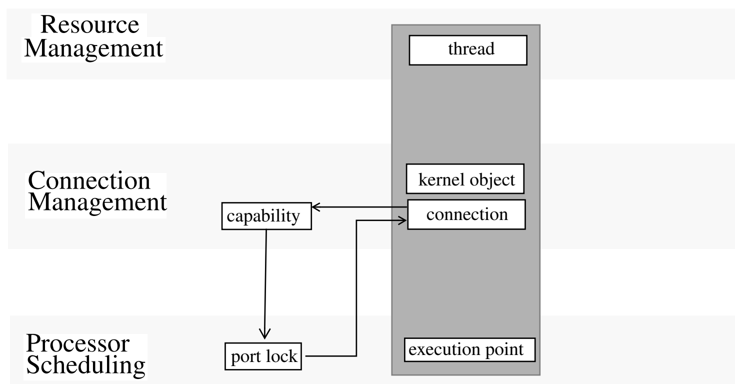


Figure 4 Thread object and components, simplified

thread_get_state system call retrieves the target thread's user computation state — register contents, program counter and user stack pointer values, and the like.)

Phase I

When invoking a MK++ system call, a user thread generates a hardware exception that causes the thread begin executing in kernel mode. To be more specific, the hardware saves some amount of the thread's user state, then transfers the thread's control to the Processor Scheduling subsystem. This subsystem saves the rest of the thread's user state and resets the CPU's registers to branch to the kernel's trap dispatcher, which is implemented in the Kernel Interface subsystem.

The diagram in Figure 5 picks up the story from there, but before the narrative, the diagram's conventions require some comment. The diagram attempts, within the confines of a static representation, to describe the dynamic nature of the Phase I operations as kernel subsystems are invoked to locate and secure sole access to the target connection. Within the diagram, rectangular items represent objects and related data structures and oval shaped items represent kernel routines and object methods. Boldfaced items and the arrows associated with them indicate points where a new kernel routine or method is invoked. For example, the figure's first "frame," depicts the Kernel Interface subsystem's **user_trap_dispatch** routine invoking the **mach_msg_trap** routine which is implemented in and exported by the Resource Management subsystem. Similarly, the second frame depicts **mach_msg_trap**'s invocation of the **get_port** method associated with the *user port name* (the user's representation of the target thread object).

The following describes, frame by frame, the operations that occur within Phase I:

A. The kernel's trap dispatcher, implemented within the Kernel Interface subsystem, determines that the current trap is the result of a system call and invokes **mach_msg_trap** within the Resource Management subsystem.

B. After copying the user's message buffer into a kernel buffer to determine the user port name of the system call's target (the copyin operation is not shown), **mach_msg_trap** invokes the User Port Name Space subsystem's **get_port** routine on the user port name entry that corresponds to the target's user port name.

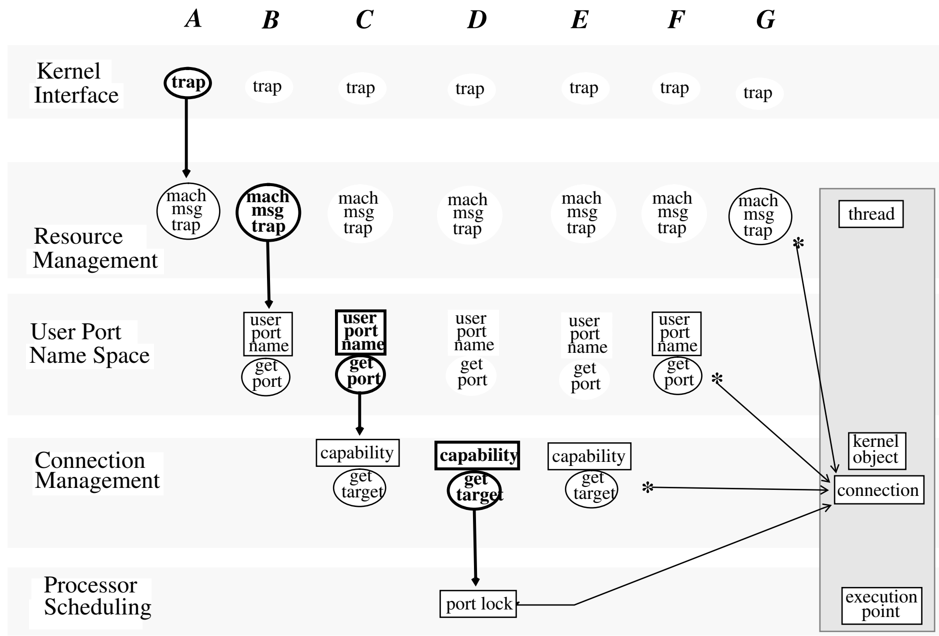


Figure 5 Kernel Object Invocation: Phase I

C. The **get_port** routine identifies the capability that underlies the user port name entry and invokes the *capability* class's **get_send_target** method on that capability; control passes to the Connection Management subsystem.

D. The **get_send_target** method invokes the Process Scheduling subsystem to validate the capability and lock the target's port_lock to secure sole access to the target object. and this pointer is the ultimate

E. The port_lock contains a pointer to the target's connection object; the Process Scheduling subsystem passes this pointer to the Connection Management subsystem when returning control to that subsystem.

F. The pointer and control are passed to the User Port Name Space subsystem.

G. The pointer and control are passed to the Resource Management subsystem, and Phase I is complete.

Phase II

With a pointer to the connection in hand, **mach_msg_trap** can send the user's message to the target object, thereby invoking the requested operation on that object. Figure 6 illustrates the operations that are conducted in Phase II. Note that a subset of the items depicted in Figure 6 are enclosed in shaded rectangles; items enclosed within the same rectangle are components of the same object, in this case the thread that is the target of the system call.

The following describes, frame by frame, the operations that occur within Phase II:

A. **mach_msg_trap** invokes the target's **send** method.

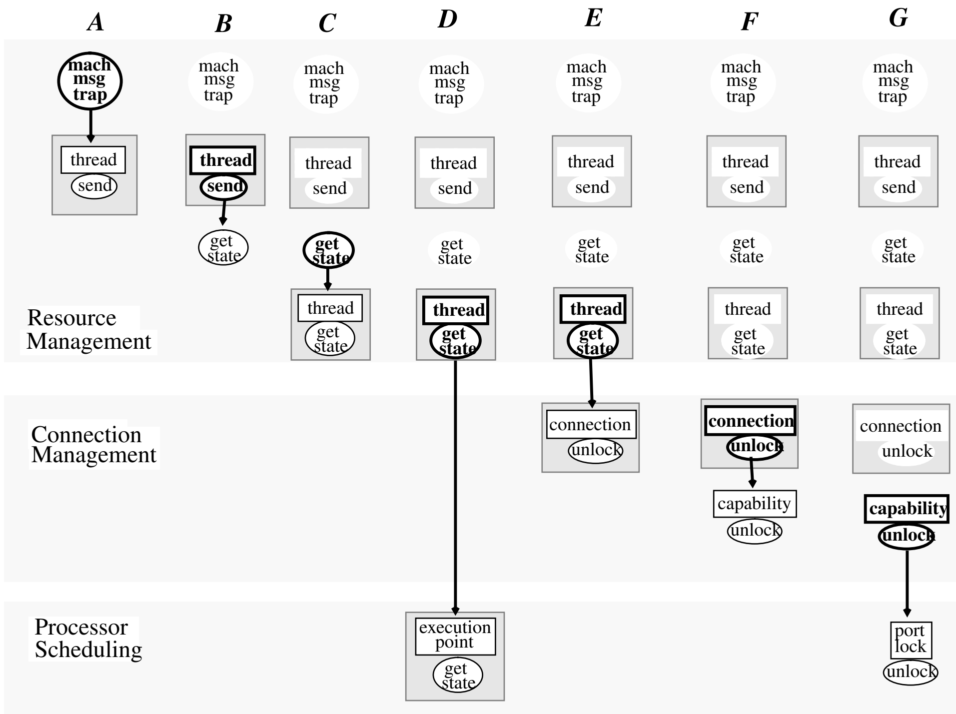


Figure 6 **Kernel Object Invocation: Phase II**

B. The **send** method invokes the kernel's thread server facility which is implemented in the Resource Management subsystem. The thread server receives and analyzes the message, then invokes its own **get_thread_state** routine on the target connection.

C. This routine further processes the message before invoking the *thread* class's **get_state** method.

D. The **get_state** method invokes the Processor Scheduler subsystem to get the thread's user state.

E. The **get_state** method invokes the Connection Management subsystem to unlock the connection.

F. The *connection*::**unlock** method uses the connection's pointer to the capability to invoke the *capability* class's **unlock** method.

G. The *capability* class's **unlock** method invokes the Process Scheduling subsystem to unlock the port_lock.

User Port Composition

The fundamental difference between user port connections and kernel object connections is rooted in the fact that messages can be queued on a given user port whereas kernel objects do not support message queueing. All functionality associated with the queueing of messages is implemented in the Transfer Management subsystem and so a significant amount of user port state is maintained within this subsystem.

Figure 7 shows a simplified representation of a user port's composition. As shown in the figure, a user port object encapsulates a user_port object and connection object, and a transfer_route object. Each user port references a transfer_queue object that functions as the port's message queue. It is important to note that a user port object does not encapsulate the transfer_queue; the separation between a user port and its message queue is in place to support the *port set* abstraction.

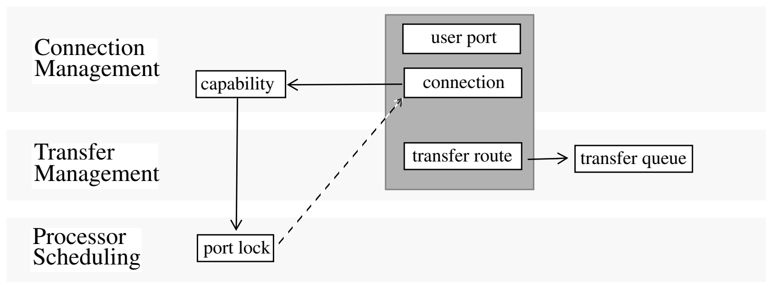


Figure 7 **User Port Composition**

The port set abstraction allows a task's threads to receive messages from multiple ports through a single name in the task's user port name space. If, for example, a task maintains a port set with three members, it can receive messages sent to either of the three ports through the port set's name.

Figure 8 shows the composition of a port set that contains three members. As shown in the figure, the port_set object encapsulates a transfer_queue object, and each port in the set references this transfer_queue. In other words, the three ports share the same message queue.

Given the fact that a user port's message queue is implemented by the transfer_queue abstraction, what is the purpose of a user port's transfer_route component? The transfer_route component has two major functions: it implements the IPC facility's message sequence number (ordering) semantic and it provides a mechanism that allows code running at interrupt level to send messages directly to user ports.

Message Sequencing

Messages are always stamped with a port-specific sequence number when they are received. This enables multiple threads that are receiving messages on the same port to reconstruct the order in which messages were sent. Messages received from port sets are stamped with a sequence number *and* the identity of the port to which the message was sent. Port sequence number processing is performed by the port's transfer_route object.

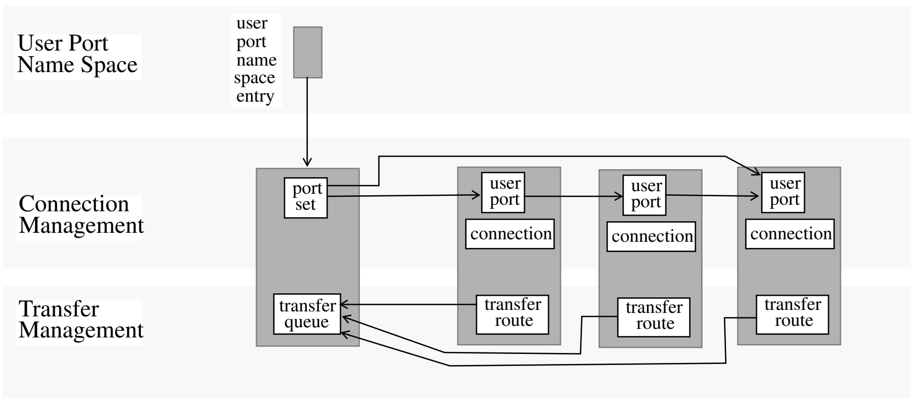


Figure 8 **Composition of a Port Set with Three Members**

Port Access Synchronization and Interrupt-Level Code

The kernel exports interfaces to the system's I/O devices. From a semantic point of view, these interfaces behave as though they are synchronous RPCs; the user invokes an I/O operation by sending a message to the port representing the target device, then waits for the kernel's device services to indicate the operation's completion by sending a reply message on the reply port the user provided when it invoked the operation.

The MK++ kernel allows interrupt-level code to send messages to user ports directly. Although the transfer_queue is protected by a spin lock and can therefore be accessed from interrupt level, interrupt level code cannot locate the transfer_queue in the normal way (which involves acquiring the user port's connection lock). Instead, interrupt level code (and other kernel code as well) locates the target transfer_queue via objects called *connection agents.*

Figure 9 shows a user port that has a single associated connection_agent. A connection agent does not need to lock the user port connection to locate the associated transfer route (and target transfer queue) because the agent is directly linked against the transfer route (under protection of the transfer queue spin lock). When the user port is destroyed, the attachment of the connection agents is changed so they send their messages through the "dead letter" transfer route. In this way, a connection agent will always handle the messages sent through it, arranging for destruction if the target port is destroyed.

Sending Messages to User Ports

Figure 10 provides a frame-by-frame description of the series of object invocations required to send a message to a user port. Assume that the user port's connection has already been locked and that **mach_msg_trap** has a pointer to the user port target of the send operation. The operation proceeds as follows:

A. **mach_msg_trap** invokes Connection Management subsystem via the user port's **send** method.

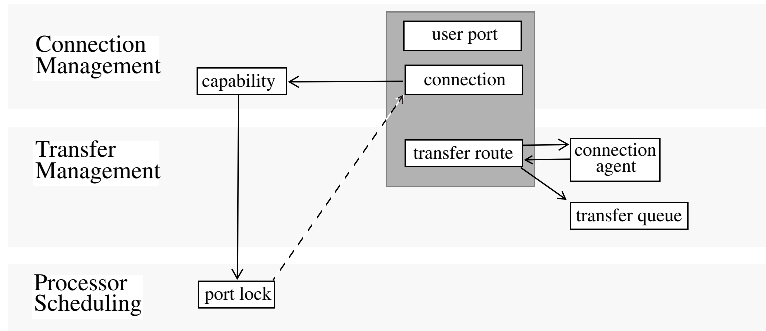


Figure 9 **User Port with an Active Connection Agent**

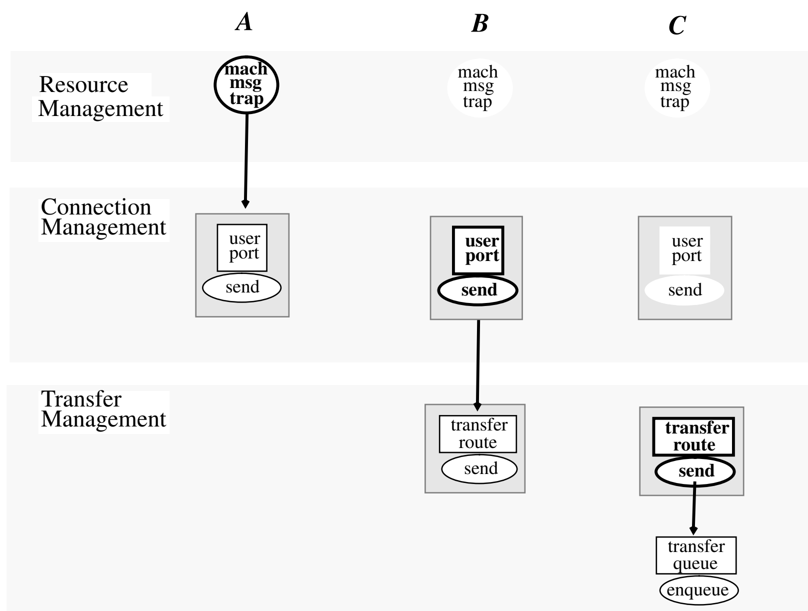


Figure 10 **Sending a Message to a User Port**

B. The user port's **send** method invokes the Transfer Management subsystem by calling the transfer route's **send** method.

C. The transfer route's **send** method invokes the transfer queue's **enqueue** method. If there is a thread waiting to receive a message from the transfer queue, the **enqueue** method hands the message to the thread and invokes the Processor Scheduling subsystem to wake the thread. Otherwise, the **enqueue** method places the message on the transfer queue.

Connections, Kernel Memory, and Space Accounting

A given connection represents either a first class kernel object or a user port; in either guise, the connection represents a consumer of memory within the kernel's address space. For example, the connection's state is maintained in a data structure that was allocated as needed from the kernel's reserve of non-pageable memory.

Non-pageable memory is a system resource that can potentially be subject to abuse by user space entities. A user task, for example, may indiscriminately allocate kernel data structures and I/O buffers. In itself, the MK++ kernel is not capable of deciding whether or not a given entity is over-allocating memory; decisions of this nature must be made in the context of a policy and the kernel does not implement such policy. The kernel does provide mechanisms for tracking kernel memory consumption and can, when instructed by a policy module, terminate offending entities and forcibly reclaim memory they have pirated.

With respect to the management of the kernel's memory, there are three types of data structures: kernel object structures, non-kernel object kernel structures, and space account structures.

Kernel object structures implement user-visible first class kernel objects such as tasks, threads, and device objects. These are "stable" structures that "belong" to the same entity (space account) throughout their existence. These structures are referred to as *space sources* because they provide a convenient means for recording kernel memory consumption.

Many composite types are associated with non-kernel object kernel structures that encapsulate state that users manipulate indirectly (and referenced as *random* structures for lack of a better term). For example, each task manages its user port name space via a port name space object. This object cannot be directly accessed from the user-level; the user manipulates a given port name space object through the task that owns it. Each random structure is always associated with a space source structure and the kernel charges a random structure's memory consumption against the space source with which it is associated. Using the previous example, the space consumed by a given user port name space object is maintained by the space source encapsulated in the task that owns the user port name space object.

Random structures can be categorized according to whether or not the structure has a fixed size across all instances and whether or not the structure's space consumption can be charged to different space sources as it binds itself first to one source, then another. The data structure that implements the IPC message abstraction is an example where both the structure's size and its owning space source are variable. Message sizes depend on the amount of data contained in each message's buffer. Also, a message changes owners as it is transferred between tasks; the kernel initially charges the message's consumption against the task sending the message, then charges the message to the user port when placing the message on the destination message queue.

IPC Abstractions

Message Rights

When a right to a port is transmitted between tasks via a message, it is transmitted in the form of a message_right object. Each message_right object specifies a mode of access and contains a capability to a port. These objects are not exposed to the user level, but their existence is implicit in the passing of port rights in messages.

Messages

Messages are important user-visible abstractions, especially since they are frequently used to transmit port rights between tasks. The basis for a message is the transfer_item implemented by Transfer Management. The contents of a message are considered to be free-form data so a decomposition of a message is not shown.

Task and Thread Decomposition

The thread object presents a user task with access to the kernel's thread manipulation interfaces. Figure 11 shows the composition of the thread object.

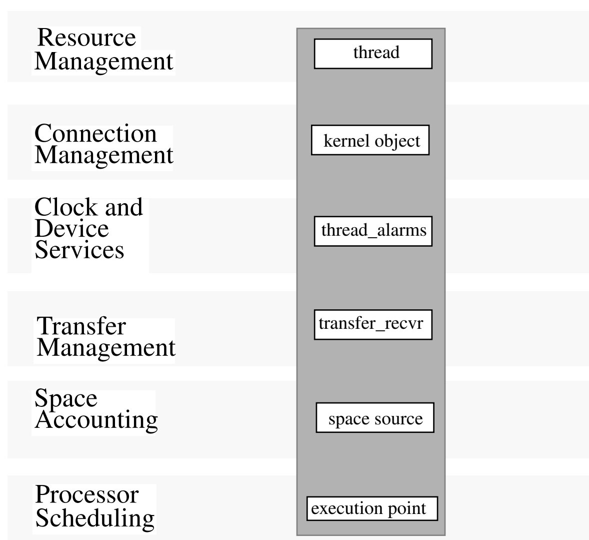


Figure 11 Thread object composition

The task object provides user tasks with access to the kernel's task manipulation operations. Figure 12 shows the decomposition of the task object.

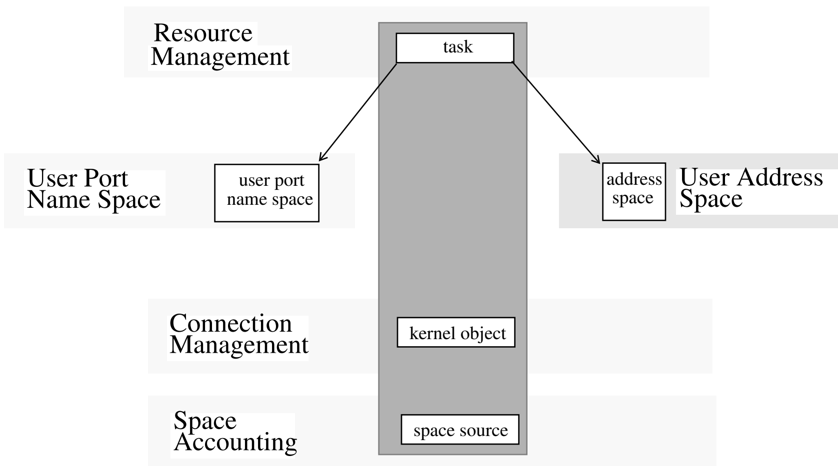


Figure 12 **Task object composition**

Ledger Decomposition

The ledger object provides user level tasks with access to the kernel's resource accounting facility. Figure 13 shows the decomposition of the ledger object.

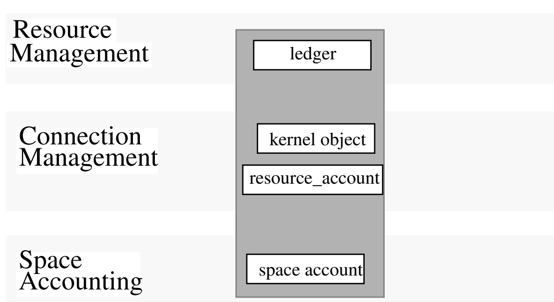


Figure 13 **Ledger object composition**

Host Decomposition

The host object allows user tasks to access information about the host system. Only privileged tasks may modify the data represented by the host object. Figure 14 presents the decomposition of the host object.

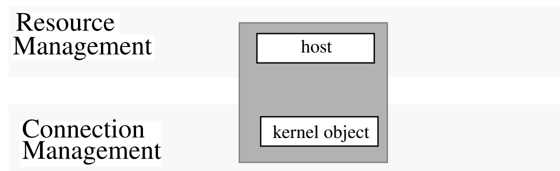


Figure 14 **Host object composition**

Virtual Memory Management

The Virtual Memory service constitutes one of the more complicated aspects of the kernel. This section provides a description of the relationships between the major objects that are involved in the management of virtual memory.

Address Spaces and Regions

For any untrusted user task, the only visible abstraction associated with the kernel's Virtual Memory services is the task's *address space*, which is represented within the kernel by an address_space object.

A task's address space contains *regions* of allocated memory. A region may contain *named data* or it may contain *anonymous data*. A task may map *named data* into its address space via the **vm_map** interface and it may allocate regions of *anonymous data* via the **vm_allocate** interface.

Within the kernel each region of named data is represented by a named_virtual_region object and each region of anonymous data is represented by an anonymous_virtual_region object. Each of these objects contain the following state with respect to the data:

- The task's access rights to the data.
- The region's inheritance characteristics; a child task may inherit a copy of the data, or may inherit shared access to the data, or may not inherit access to the data at all.

The User Address Space subsystem implements and manages address_space objects and both types of virtual region objects; note that address_space objects are exported, but virtual region objects are not exported.

Virtual Memory Objects and Virtual Memory Extents

Each region within an address space maps a range of the address space to a set of virtual pages. Within the kernel, each set of pages is managed by a *virtual memory object*. There are two types of virtual memory objects: named_virtual_memory objects, and anonymous_virtual_memory objects.

A region is not mapped directly to a virtual memory object. Instead it is mapped to a *virtual extent*. A virtual extent represents a set of contiguous virtual pages within a virtual memory object. An extent may map all of the pages within the virtual memory object, or it may map a subset of those pages.

The virtual extent abstraction enables multiple tasks to map the same virtual memory object into each of their address spaces. A separate virtual extent represents each mapping; if two tasks map to the same virtual memory object, the kernel uses two separate extents to represent the mappings.

Virtual extents that map to the same virtual memory object may map to the same set of pages or overlapping sets of pages. In Figure 15, three extents map to the same virtual memory object. The virtual memory object represents four virtual pages. Extent *A* maps to all four pages; extents *B* and *C* map to the last two pages.

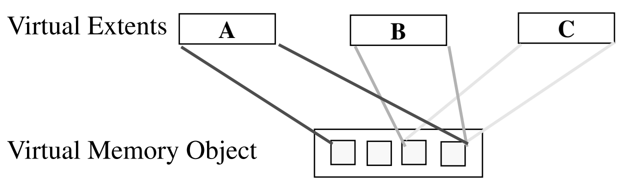


Figure 15 Extents Mapping a Virtual Memory Object

As might be expected, an extent that maps to a named memory object is a named_virtual_extent and an extent that maps to an anonymous memory object is an anonymous_virtual_extent. The kernel's Memory Extent Management subsystem implements and manages the virtual memory and virtual extent objects.

Resident Memory Objects

A given virtual page within a virtual memory object is either in resident memory or in secondary storage. A page that is in resident memory is said to be *cached* in resident memory.

The kernel uses a *resident memory object* to represent the set of a virtual object's pages that are currently in resident memory. There are two types of resident memory objects: named_resident_memory and anonymous_resident_memory objects. Both types are implemented and managed by the Resident Memory Management subsystem.

Management of Named Data

Named data is managed by the system's memory managers. A memory manager manages its named data via the *memory object* abstract. For example, in the case of a memory manager that implements access to data files, each file would constitute a memory object.

It is important to note the difference between memory objects and virtual memory objects. A memory object represents data as it exists in secondary storage; a virtual memory object is the kernel's representation of data that is mapped by one or more tasks.

Memory Object Representatives

In order to map a memory object into its address space, a task must be able to identify the memory object. Memory managers make such identification possible by supplying a *memory object representative* for each memory object. A memory object representative is a port that the memory manager uses to handle mapping requests on the memory object.

When a task issues a call to **vm_map** to map a particular memory object, it supplies a capability to the object's memory object representative; in processing the **vm_map** call, the kernel sends a message to the memory manager via the memory object representative port. The sending of this message begins a series of events that is referred to as *memory object initialization*.

Note that memory object initialization is performed only on the initial mapping call. If a task attempts to map a memory object that is already mapped by another task, the memory object is already initialized and does not need to be initialized again.

Kernel and Memory Manager Interactions

If a task attempts to reference a page of named data that is not in resident memory, it must be paged in from secondary storage as follows: the kernel sends a page-in request to the memory manager and the memory manager responds by sending the page to the kernel. The kernel caches the page in resident memory and makes it available to the task.

Later, if the page is modified and becomes the target of the kernel's page replacement algorithm, the kernel sends a pageout request to the memory manager. The memory manager responds by paging out the data and informing the kernel that the page has been so processed. The kernel can then reallocate the physical page.

These interactions between the kernel and the memory manager are performed via the IPC service and are conducted via two ports: the kernel sends its paging requests to the memory manager via the *abstract memory object* and the memory manager sends its responses to the kernel via the *memory cache object*.

The memory manager also sends cache management messages to the kernel via the memory cache object; such a message may cause the kernel to change a given page's protection attributes, for example.

Named Data and Resident Memory Management

As mentioned above, a memory object's pages that are currently cached in resident memory are represented by a named_resident_memory object. In addition to tracking the resident pages, this object contains a connection agent that is bound to the abstract memory object port. All pagein and pageout requests are sent to the memory manager via this connection agent.

Management of Anonymous Data

Unlike named data, which usually persists in secondary storage before and after it is mapped into a task's address space, anonymous data is created in resident memory as the task executes and is destroyed when the task is destroyed. This is not strictly true; if a child task inherits a region of anonymous memory from its parent, the data will not be destroyed when the parent is destroyed if the child continues to exist.

A given page of anonymous memory stays in resident memory unless it becomes subject to the kernel's page replacement algorithm. Should this occur, the page is paged out to the kernel's *anonymous paging space*. This paging space, which is managed by the system's default pager, is represented within the kernel by the virtual_paging object. The kernel exports this object for the default memory manager's benefit.

The virtual_paging object contains the kernel's resident_paging space object. This object maintains the kernel's map of anonymous secondary storage. The kernel uses this map to locate specific pages in secondary storage.

Address Space Decomposition

Figure 1 shows the decomposition of an address_space object with several mapped regions. The diagram does not show the objects that underlie the mapped extent objects.

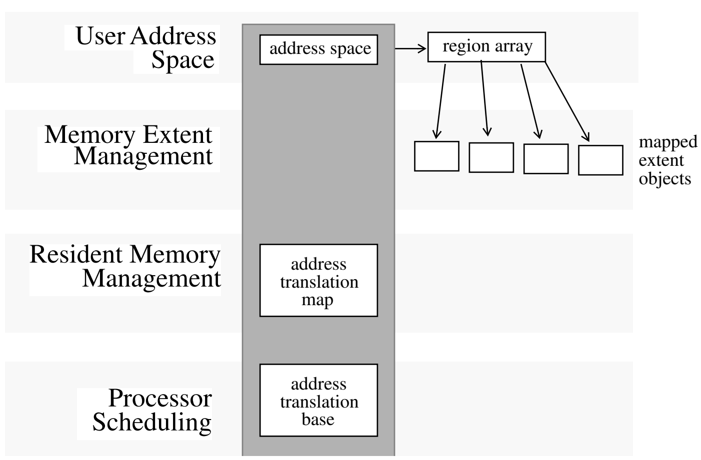


Figure 1 User address space composition

Physical Devices

The device master port is one of the several distinct privileged ports that name the host object. Via this port, used for the **device_open** call, the kernel locates the appropriate device_control object. Each of these objects represents a specific device that is configured into the system.

Each instance of a device is represented by a hw_device object exported by Clock and Device Services, as well as structures within Device Management. Each opening of a device is represented by a kernel_device object. Figure 16 shows the decomposition of a kernel_device object. Each kernel_device references the actual hw_device object in the Clock and Device Services subsystem.

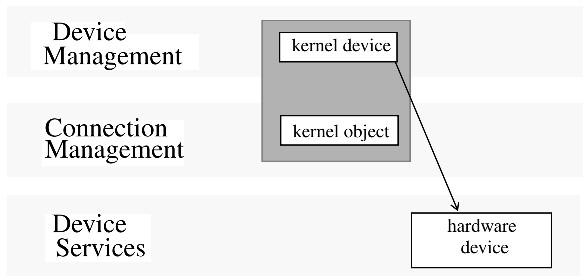


Figure 16 Kernel device object composition

Clock Service

The kernel_clock object provides user tasks access to the kernel's clock facility. Figure 17 shows the decomposition of a kernel_clock object.

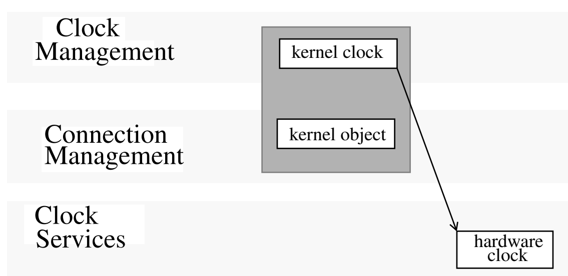


Figure 17 Kernel clock object composition

The kernel_alarm object represents an alarm set by a user task. Figure 18 shows the decomposition of a kernel_alarm object.

Processor Set

The processor_set object provides user tasks with access to the kernel's processor set facility. Figure 19 shows the decomposition of the processor_set object.

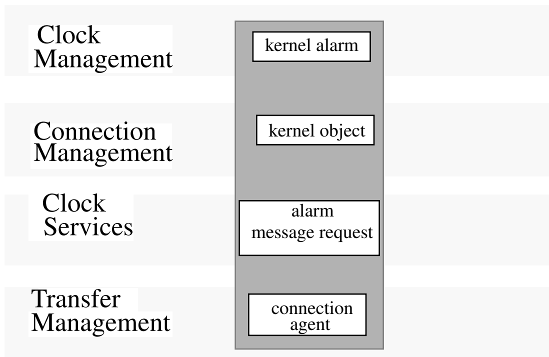


Figure 18 Kernel alarm object composition

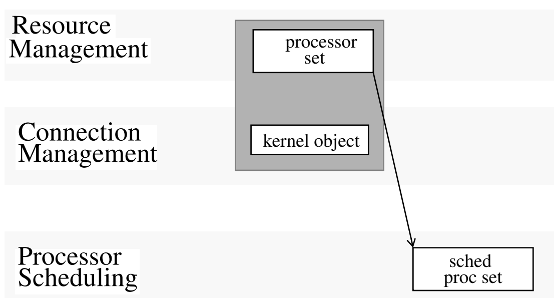


Figure 19 Processor set object composition

Entering the Kernel

There are two general mechanisms that handle transfer of control into the kernel: the trap mechanism employed by user threads, and the hardware interrupt mechanism employed by the system's physical devices.

Hardware Interrupts

All hardware interrupt handlers are contained in the kernel's Clock and Device Services subsystem. During system initialization, this subsystem calls the Processor Scheduling subsystem to initialize that subsystem's table of interrupt vectors. Each vector matches an interrupt type with the appropriate interrupt handler.

When an interrupt occurs (assuming that the CPU is not currently masking the interrupt type), the hardware transfers control to the Processor Scheduling's interrupt code.

Together, Figure 20 and the following list of steps outlines how the interrupt mechanism transfers control to and from the appropriate interrupt handlers.

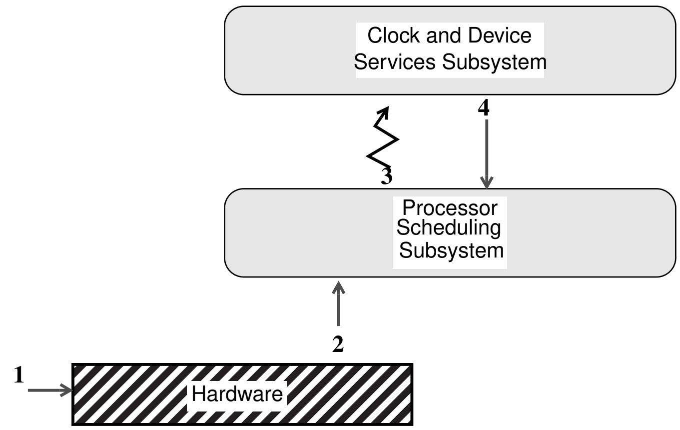


Figure 20 **Interrupt Processing**

1. A device interrupt occurs, raising a predefined interrupt condition.
2. If the given interrupt is not currently being masked, the hardware saves some amount of the current thread's user state, then invokes the Processor Scheduling subsystem's interrupt handler.
3. The Processor Scheduling subsystem saves the current CPU state, interprets the interrupt type, locates the appropriate interrupt handler within its internal table, and resets the CPU's registers to branch to the interrupt handler. This transfers control to the Clock and Device Services Subsystem, as indicated by the zig-zag line.
4. When the interrupt handling routine completes its operation, it calls the Processor Scheduling subsystem to resume the previous level of execution.

The Trap Mechanism

Generating a trap is a user thread's only means of entering the kernel. All MK++ system calls are invoked via trap, and all exceptions and page faults enter the kernel in this manner. In the following subsection, the focus is on how the trap mechanism handles system call invocation. In the subsections that immediately follow, how the trap mechanism handles exceptions and page faults will be discussed.

No matter what the circumstances, any user entry to the kernel is an entry to the Kernel Interface subsystem. During system initialization, this subsystem calls the Processor Scheduling subsystem to initialize that subsystem's exception vector.

System Call Trapping

Together, Figure 21 and the following list of steps describe how the trap mechanism handles system call invocation.

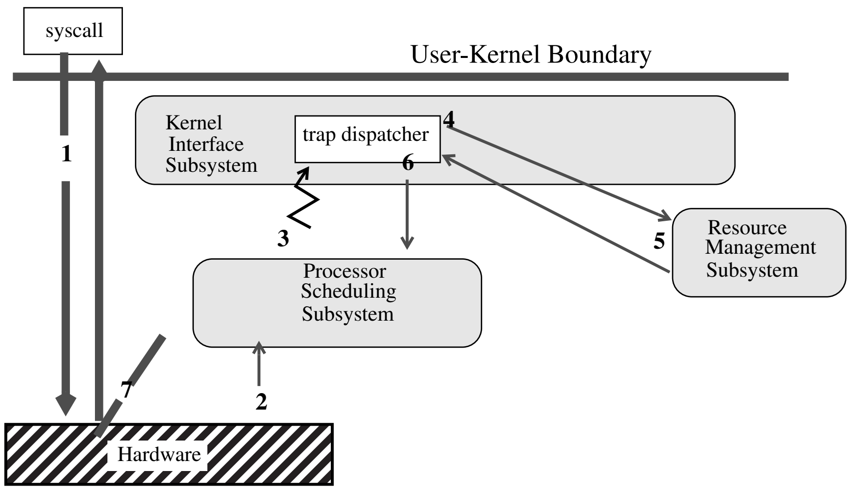


Figure 21 **Trap Mechanism**

1. The current thread generates a trap via **syscall**, thereby invoking the hardware.
2. The hardware saves some amount of the thread's user state, then invokes the Processor Scheduling subsystem.
3. The Processor Scheduling subsystem completes saving the thread's user state, and resets the CPU's registers to branch to the trap dispatcher. This transfers control to the Kernel Interface subsystem, as indicated by the zig-zag line.
4. The trap dispatcher decodes the machine state to determine that it is dealing with a **syscall** trap. On the basis of the accompanying syscall number, the trap dispatcher copies into the kernel's address space any requisite parameters. The trap dispatcher then calls the appropriate trap routine in the Resource Management subsystem. For example, if the system call is a kernel RPC, the Kernel Interface subsystem calls **mach_msg_trap**.
5. When the trap routine returns, the Kernel Interface subsystem copies out to the user's address space the return value of the operation.
6. The Kernel Interface subsystem calls the Processor Scheduling subsystem to prepare the thread to resume execution in user mode.
7. The Processor Scheduling subsystem checks to see if there have been any thread management requests (*suspend, terminate, etc.)* made against the thread while it was executing in kernel mode. If no such requests are pending, the Processor Scheduling subsystem restores the thread's user state and the thread resumes executing in user mode.
8. Otherwise, the Processor Scheduling subsystem returns control (to the Kernel Interface subsystem) and that subsystem initiates the operations necessary to handle the suspend or thread terminate request. (This step is not shown in the figure.)

CHAPTER 3

Kernel Interface Subsystem Design

Overview

The Kernel Interface subsystem occupies the highest layer in the Kernel's subsystem hierarchy and implements the boundary between user mode execution and kernel mode execution. This subsystem manages all user-initiated kernel invocations, including explicit invocations, such as system call traps, and implicit invocations, such as page faults.

The subsystem implements the top level handler for all kernel entries. The subsystem itself implements relatively few complete operations. Instead, it invokes one or more lower subsystems to perform the operations required with a given kernel invocation.

The Kernel Interface subsystem also encapsulates the kernel's exit point to user space; all user-initiated kernel invocations exit back to user space via this subsystem.

An ancillary task of this subsystem is to create the Primitive Bootstrap task and bootstrap thread, and initiate that thread's execution.

Kernel Mode Entry

A thread executing in user mode enters kernel mode via one of the following mechanisms:

- Explicit trap instructions, such as those generated by the invocation of the kernel interface.
- Page fault and other memory faults.
- General exceptions caused by the user instruction stream such as an overflow exception or a divide-by-zero exception.

Transfer of Control to the Kernel Interface Subsystem

The transfer of control from user mode to kernel mode necessarily involves the hardware; all hardware-related processing of such events is performed within the Processor Scheduling subsystem, including the transition to and from kernel mode.

The Processor Scheduling subsystem contains an *exception vector table* that matches specific kernel entry points to each defined hardware exception. The Kernel Interface subsystem registers a single entry point during system start-up by invoking the appropriate method exported by the Processor Scheduling subsystem. This entry point is **user_trap_dispatch**, the subsystem's handler for all logical kernel entry points.

When a user mode thread invokes any of the kernel entry mechanisms, the hardware reports the event to the Processor Scheduling subsystem. The Processor Scheduling subsystem performs machine-dependent processing (such as saving registers) and uses the exception vector table to transfer control to **user_trap_dispatch**.

Trap Dispatching

When transferring control to **user_trap_dispatch**, the Processor Scheduling subsystem passes along information that allows **user_trap_dispatch** to determine the nature of the kernel entry; **user_trap_dispatch** uses this information to determine what specific Kernel Interface routine to dispatch. For example, if the current thread entered the kernel by invoking an RPC on a kernel object, **user_trap_dispatch** invokes the Resource Management **mach_msg_trap** routine to process the RPC.

In processing a given entry into the kernel, the Kernel Interface subsystem may refine the entry's characterization as reported by the Processor Scheduling subsystem. For example, in processing what was characterized as a page fault, the Kernel Interface subsystem, after analyzing the entry by invoking the User Address Space subsystem, may determine that the fault is actually a memory access exception.

The system hardware, in concert with the Processor Scheduling subsystem, vectors to a pre-announced entry point within this subsystem to handle externally generated events:

- **user_trap_dispatch** — The entry point to which the processor transfers when any exception in user space occurs. The **user_trap_dispatch** entry point analyzes the trap, invokes the appropriate internal routine. When all kernel processing is finished, **user_trap_dispatch** invokes the Processor Scheduling subsystem to transfer control back to user mode.

The subsystem defines a special *null* trap that the kernel uses internally to handle the following circumstance: the current thread was executing in user mode and an interrupt forced the processor to kernel mode; in returning from the interrupt, the kernel determines that a high-level thread Scheduling request is pending on the thread.

In this situation, the scheduler fabricates a *null* trap when the processor next runs in the context of that thread. The Resource Management subsystem responds to the null trap by trying to return from the kernel, possibly invoking the higher level thread Scheduling primitives if necessary.

Exiting from Kernel Mode

When the kernel completes the processing of a kernel invocation, the Kernel Interface subsystem invokes the Processor Scheduler subsystem to transfer control to user mode. Before performing the operations necessary to transfer control to user mode, the Processor Scheduling subsystem first determines whether or not it is appropriate to continue the thread's execution. The Processor Scheduling subsystem will not continue a thread's execution if the thread has been preempted by the scheduler, or if there is a high-level thread Scheduling request such as *thread suspend* or *thread terminate* pending on the thread.

If the thread is to be preempted, the Processor Scheduling subsystem performs operations internally to change the processor's context to that of the preempting thread. If a high-level thread Scheduling request is pending, the Processor Scheduling subsystem returns with a flag that causes the Kernel Interface subsystem to carry out the request.

If the Processor Scheduling subsystem determines that it is appropriate to continue executing the current thread in user mode, it restores the thread's user state and performs the hardware-related operations required to change the CPU's processing state to user mode.

Subsystem Interface

As the top layer within the kernel, this subsystem is not called per se by any other part of the kernel.

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

As the user interface to the facilities of the kernel, this subsystem uses the services of virtually all of the other kernel subsystems:

- Resource Management Objects for system traps, task, thread, processor set and ledger calls.
- User Port Name Space for port name to right translation.
- User Address Space for handling user page faults.
- Device Management for device calls.
- Clock Management for clock and timer calls.
- Memory Extent Management for memory object calls.
- Connection Management for right to port translation.
- Clock and Device Services to process device and clock interrupts.
- Resident Memory Management for memory allocation.

- Transfer Management for enqueing and transfer of user messages and support of exception messages.
- Space Accounting as part of message construction.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

Although all kernel subsystems depend on physical protection from user mode tasks, the Kernel Interface subsystem is most directly aware and dependent on this protection.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (messages).
- Supports architectural requirements for trust (process isolation).
- Implements an external interface to protection-critical code (kernel interface service).
- Is expected by other TCB components to function correctly.

Functional Decomposition

This subsystem consists of the top level handler to which the hardware and Processor Scheduling subsystem vector control when the kernel is entered, and various fault handling routines for faults occurring in user context.

For convenience, the *kernel_interface* class that represents the subsystem derives from Processor Scheduling's *exception_vector* class.

Concurrency Issues

Interaction with High-Level Thread Scheduling

The Kernel Interface subsystem defines both the point of entry and the point of exit for user threads passing into and out of the kernel. With help from the Processor Scheduling subsystem, the Kernel Interface subsystem is responsible for initiating the top level implementation of operations affecting the state of the overall thread, including thread suspension and termination. This subsystem invokes the Resource Management subsystem to initiate operations on a thread.

Each thread data structure contains a *think* flag that the kernel uses to inform the thread about pending thread management requests. When a thread attempts to exit from the kernel, the Processor Scheduling subsystem checks the thread's *think* flag to determine if a high level thread management request is pending.

A thread's *think* flag is protected by the thread's lock. The thread (while in the scheduler) examines this flag while holding no lock (although the fact that the thread is still in kernel mode protects the thread's existence); the scheduler is responsible for assuring that when it sets this flag, the target thread will run in such a way that it sees it, as discussed under the Processor Scheduling subsystem. If the thread's *think* flag is set, attempting to return from the kernel fails. The thread then calls *thread*::**examine_state** which, after obtaining the thread lock, determines what thread management operation to perform, returning when performed (unless the thread is terminated) with the *think* flag reset.

Interaction with Memory Management

In general, the kernel guarantees that once a user thread enters the kernel, the requested operation will run to completion. This implies that the operation will not fail due to a lack of memory.

To support this guarantee, the Kernel Interface subsystem always checks the low memory flag exported by the Resident Memory Management subsystem before processing a given kernel entry request. If this flag is set, the Kernel Interface subsystem invokes Resident Memory to reserve memory, possibly delaying the requested operation until there is enough memory to guarantee that the operation will run to completion.

Subsystem Initialization

Initialization of this subsystem consists of establishing the kernel entry point vector and creating and launching the bootstrap task.

Languages Employed

The Processor Scheduling subsystem directly vectors to entry points within this subsystem which are machine (largely stack and fault frame) dependent and therefore implemented in assembly language. The rest of this subsystem is implemented in C++.

CHAPTER 4 Resource Management Objects Subsystem Design

Overview

The Resource Management Objects subsystem provides the kernel's high level resource management facilities. This subsystem implements the top-level view of the following closely related kernel objects involved with governing resources:

- Host
- Processor set
- Ledgers
- Tasks
- Threads

By virtue of its exposing these objects to the user, the Resource Management subsystem exports the user-visible interfaces associated with the objects. Note, however, that for the most part, the actual operations that underlie these interfaces are implemented by subsystems in lower layers.

This subsystem also implements system call traps and also performs all user-visible exception processing (the sending of an exception message and waiting for a response).

All messages intended for a user but never received ("dead" letters) are handled by the dead letter office, running in this subsystem, which knows how to destroy any rights (including possible port destruction, if a receive right) and destroy any out-of-line regions.

Host

There is a single host object that represents the kernel itself. It is through this kernel object that access to other kernel objects is initially obtained. For example, ledgers which track dynamically allocated kernel memory are created via an operation on the host object. Operations on the host object also supply basic system ports such as the device master and security ports. Lastly, the host object is invoked to retrieve basic system configuration information such as the system page size.

Processor Sets

The processor set object is the abstraction by which multiple physical processors are configured as a single virtual processor which implements the scheduling policy for its physical processors. While the nature of the processor set interfaces allows for processor sets to be dynamically created and destroyed, the initial architecture provides a single processor set kernel object to which all physical processors are assigned. Additionally all tasks and threads are bound to this processor set. The port for this default processor set is obtained via an operation on the host object.

Ledgers

The kernel tracks the amount of dynamically allocated memory by means of ledger objects. These objects, which are themselves dynamically allocated, occupy the paramount position in the hierarchy of the kernel's dynamically allocated objects. For example, in order to create a new task, a user thread must identify the specific ledger that will be used to account for the resources consumed by the new task.

Once allocated, a dynamically allocated object is said to be *drawn* from its ledger. A ledger is capable of destroying any of the objects that have been drawn from it. In order to provide this function, the Resource Management Objects subsystem must occupy a position above all other subsystems that implement dynamically allocated kernel objects.

Note that when a ledger is destroyed, all objects that are currently drawn from the ledger must be destroyed as well. The facility that manages this wholesale destruction is implemented within the Resource Management Objects subsystem.

Also note that the kernel's space management policy facility is implemented within this subsystem. This facility periodically scans the kernel's list of active ledgers and determines if any of them have over allocated memory. The facility is capable of reclaiming over allocated memory.

Tasks and Threads

The top level view of tasks and threads is defined here. This is for several reasons:

- Tasks are represented as resource containers.
- Tasks and threads are composite objects and their top-level representations must be defined above the subsystems that implement their components.

- A thread cannot be destroyed while executing in the kernel; consequently, thread termination, task management, and ledger management are intertwined and must be managed within the same subsystem.

Much of the activity implemented by this subsystem concerns the management of the linkage between related task and thread objects. These linkages are maintained in order to support the kernel interfaces that report on these linkages and also to coordinate the destruction of child objects with their parent objects. For example, in order to destroy a task, the kernel must first destroy its threads. This subsystem insures that the task and its threads are destroyed in an orderly fashion.

High level thread management primitives, such as **thread_suspend** and **thread_terminate**, are implemented within this subsystem. Note, however, that these primitives can be invoked only by the Kernel Interface subsystem, and only when that subsystem has determined that the target thread has completed its current kernel activity and would otherwise be leaving the kernel.

Subsystem Interface

The *host* class exports the following member functions which perform the action of the kernel interface by the same name:

- *host::***bootstrap_arguments**
- *host::***bootstrap_ports**
- *host::***device_open**
- *host::***host_get_boot_info**
- *host::***host_get_clock_control**
- *host::***host_get_clock_service**
- *host::***host_info**
- *host::***host_kernel_version**
- *host::***host_page_size**
- *host::***host_processor_set_priv**
- *host::***host_reboot**
- *host::***host_security_id_list**
- *host::***host_statistics**
- *host::***ledger_create**
- *host::***memory_object_establish**
- *host::***memory_object_representative**
- *host::***processor_set_default**
- *host::***task_create_security_id**
- *host::***thread_wire**

The *host* class also exports the **shutdown** which is called by the kernel interface subsystem after the kernel is initialized, to transform the initialization thread into the shutdown thread, waiting to be resumed to shutdown the kernel.

The *processor_set* class exports the following member functions which perform the action of the kernel interface by the same name:

- *processor_set*::**processor_set_info**
- *processor_set*::**processor_set_policy_control**
- *processor_set*::**processor_set_statistics**
- *processor_set*::**processor_set_tasks**
- *processor_set*::**task_set_policy**
- *processor_set*::**thread_set_policy**

The *ledger* class exports the following member functions which perform the action of the kernel interface by the same name:

- *ledger::***alarm_create**
- *ledger*::**ledger_create_port**
- *ledger*::**ledger_terminate**
- *ledger::***task_create**

In addition, the *ledger* class exports the following member functions to higher layers:

- *ledger*::**ledger** — Constructor.
- *ledger*::**~ledger** — Destructor.
- *ledger*::**bootstrap_task_create** — Create the bootstrap task.

The *task* class exports the following member functions which perform the action of the kernel interface by the same name:

- *task::***mach_port_allocate**
- *task::***mach_port_destroy**
- *task::***mach_port_extract_right**
- *task::***mach_port_get_attributes**
- *task::***mach_port_get_refs**
- *task***::mach_port_insert_right**
- *task::***mach_port_mod_refs**
- *task::***mach_port_move_member**
- *task::***mach_port_request_notification**
- *task::***mach_port_set_attributes**
- *task::***mach_port_type**
- *task::***task_get_assignment**
- *task::***task_get_exception_ports**
- *task::***task_get_special_port**

- *task::***task_info**
- *task::***task_policy**
- *task::***task_resume**
- *task::***task_sample**
- *task::***task_set_exception_ports**
- *task::***task_set_special_port**
- *task::***task_suspend**
- *task::***task_swap_exception_ports**
- *task::***task_terminate**
- *task::***task_threads**
- *task::***thread_create**
- *task::***thread_create_running**
- *task::***vm_allocate**
- *task::***vm_behavior_set**
- *task::***vm_copy**
- *task::***vm_deallocate**
- *task::***vm_inherit**
- *task::***vm_map**
- *task::***vm_protect**
- *task::***vm_read**
- *task::***vm_region**
- *task::***vm_reserve**
- *task::***vm_unreserve**
- *task::***vm_unwire**
- *task::***vm_wire**
- *task::***vm_write**

The *task* class also exports the following member functions for the use of the Kernel Interface subsystem:

- *task*::**get_address_space —** Return the task's address_space.
- *task*::**syscall_local —** Determine whether the local host handles MK++ syscall exceptions.
- *task*::**syscall_remote —** Determine whether an external task handles MK++ syscall exceptions.

The *thread* class exports the following member functions which perform the action of the kernel interface by the same name:

- *thread::***thread_exception_continue**
- *thread::***thread_exception_continue_state**
- *thread::***thread_get_assignment**
- *thread::***thread_get_exception_ports**

- *thread::***thread_get_special_port**
- *thread::***thread_get_state**
- *thread::***thread_info**
- *thread::***thread_notify**
- *thread::***thread_policy**
- *thread::***thread_resume**
- *thread::***thread_sample**
- *thread::***thread_set_exception_ports**
- *thread::***thread_set_special_port**
- *thread::***thread_set_state**
- *thread::***thread_suspend**
- *thread::***thread_swap_exception_ports**
- *thread::***thread_terminate**

The *thread* class also exports the following member functions for the use of the Kernel Interface subsystem:

- *thread*::**allocate_float_state** — Allocate floating point storage for this thread if it does not already exist.
- *thread*::**call_entry** — Arrange for the function specified to be invoked with the number of arguments specified given this execution point's machine state. Used to invoke the trap routines corresponding to user mode trap entries.
- *thread*::**current** — Return the active thread for the current CPU.
- *thread*::**examine_state** — Called when an attempt to exit kernel mode fails because the scheduler's *think* attribute was set. This call checks to see if the thread should terminate, suspend, abort or notify. If no thread management operations apply, this call will reset the *think* attribute.
- *thread*::**free_float_state** — Free this thread's floating point state storage if it exists.
- *thread*::**parent** — Return the parent task for this thread.
- *thread*::**send_exception** — Deliver the specified exception to the appropriate port.
- *thread*::**set_syscall_return** — Set syscall return value for this thread
- *thread*::**syscall_local** — Determine whether the local host handles MK++ exceptions.
- *thread*::**syscall_remote** — Determine whether an external task handles MK++ exceptions.

The *kernel_thread* class represents the highest derivation of any kernel-internal thread; it exports the following member functions:

- *kernel_thread*::**kernel_thread**
- *kernel_thread*::**~kernel_thread**

The *resource_management_objects* class (for which there is only a constructor for initialization) represents the subsystem as a whole.

This subsystem also comprises various trap handling routines that the top level handler calls to handle specific kernel entries. The trap handling functions are the following:

- **clock_sleep_trap** — User trap to delay (interruptibly) for a specified time
- **get_time_stamp_trap** — User trap to return current system time
- **mach_host_self_trap** — User trap to return the host self special port name.
- **mach_msg_trap** — User trap to send a message to another task or to invoke a kernel object.
- **mach_reply_port_trap** — User trap to create and return a new port.
- **mach_task_self_trap** — User trap to return the task self special port name.
- **mach_thread_self_trap** — User trap to return the thread self special port name.
- **thread_continue_state_trap** — User trap to resume execution of a thread from the point at which it took its last exception, optionally first setting its state and optionally waiting for a message.
- **thread_depress_abort_trap** — User trap to abort the a scheduling priority depression.
- **thread_switch_trap** — User trap to (possibly) give up the processor with optional priority depression.
- **vm_read_inline_trap** — Read a task's virtual memory returning the data in the supplied buffer.
- **vm_write_inline_trap** — Write a task's virtual memory with the data in the supplied buffer.

mach_msg_trap

The handling of messages sent to and from user space involves the following:

- The system trap handler itself, which copies user messages between user and kernel space, calling upon the User Port Name Space subsystem for right translations and the User Address Space subsystem for the handling of out-of-line data.
- For messages sent from the user to a user port, **mach_msg_trap** invokes the Connection Management and Transfer Management subsystems.
- For messages sent from the user to a kernel object, **mach_msg_trap** invokes the Connection Management subsystem to locate the target kernel object. Once found, code within the kernel object executes (within this layer) to translate the message into an invocation of a specific method upon the object. This translation is performed by code generated by the kernel Mach Interface Generator (MIG) tool when the kernel is built.
- All messages sent to a user (either by the kernel or by another user task) involve the Connection Management and Transfer Management subsystems.
- All messages intended for a user but never received ("dead" letters) are handled by the dead letter office, running in this layer, which knows how to destroy any rights (including possible port destruction, if a receive right) and destroy any out-of-line regions.

Table 1 shows the inheritance relationship for this subsystem's objects. Classes reside in the Resource Management subsystem except where annotated with another subsystem name.

Table 1 Resource Management Class Inheritance

Class	Inherits from
cached_message	message (IPC xfer)
callout_object	callout
callout_ptr	callout_data
callout_table	callout_data
harvest_object	kernel_object (IPC conn)
harvest_request	transfer_item (IPC xfer)
host	harvest_object harvest_parent
kernel_thread	message_holder
ledger	harvest_object harvest_parent resource_account (IPC conn)
message_holder	thread_alarms (Clk svc)
processor_set	kernel_object (IPC conn)
sample_object	harvest_object sample_data (Sched)
task	sample_object harvest_parent exception sched_policy (Sched)
thread	sample_object message_holder exception

Security Policy

This subsystem maintains the security ID for tasks. It is responsible for correctly setting the ID at task creation, for supplying it when requested (by the kernel interface) for tagging messages, and for supplying it as a parameter when calling the User Address Space subsystem to map a memory object.

Privileges and Capabilities

This subsystem establishes a security ID for a new task in support of the **task_create_security_id** call.

This subsystem requires the services of the following subsystems:

- User Port Name Space for task port name space calls.
- User Address Space for task address space calls.
- Connection Management for right to port translation, and for enumerating ledger resources.
- Clock and Device Services for time services.
- Resident Memory Management for memory allocation.
- Processor Scheduling for synchronization and processor set management.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (tasks).
- Provides system facilities used to represent objects and subjects (tasks).
- Provides system resources which are used to implement objects and subjects (tasks, threads).
- Supports architectural requirements for trust (process isolation).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 2 shows the object instance relationships for this subsystem.

Table 2 Resource Management Object Relationships

Object	References
exception	exception_port
host	processor_set virtual_paging (VM ext)
processor_set	task thread mutex (Sched) sched_proc_set (Sched)
sample_object	recycling_msg (IPC conn)
task	processor_set port_name_space (IPC space) address_space (VM addr) sched_policy (Sched) cpu_usage (Sched)

Table 2 Resource Management Object Relationships

Object	References
thread	processor_set task event (Sched) sched_policy (Sched)

Thread (and Related Object) Destruction

A thread cannot destroy itself because the operation takes place within the kernel and a thread cannot be destroyed while it is executing in the kernel. Consequently, a thread never invokes its own destructor. Instead it invokes a method that marks the thread for destruction and returns. Later, when the thread attempts to transition back to user mode, the Kernel Interface subsystem determines that the thread can now be destroyed and sends the thread's identity (the capability to the thread's control port) to the kernel's internal harvest_thread. This thread invokes a method on the thread that deallocates its structure.

Thread destruction complicates the destruction of task and ledger objects; for example, in order to destroy a task, the kernel must first destroy the task's threads, and in order to destroy a ledger, the kernel must first destroy all of the tasks drawn from the ledger, and of course that implies that all the threads belonging to those tasks must first be terminated.

Like threads, task and ledger objects are deallocated via the harvest_object mechanism. For example, when the kernel is invoked to destroy a task, the task's destructor marks the task for destruction, initiates the destruction of the task's threads and returns. When the last thread within the task is destroyed, the kernel sends the task's identity to the harvest_object for deallocation.

The Resource Management Objects subsystem implements the orderly destruction of threads, tasks, and ledgers via the *harvest_object* class. Note that this class derives from *kernel_object* and participates in the derivation of the *task*, *thread* and *ledger* classes. The harvest_thread deallocates a given object by invoking the object's **harvest** method.

These linkages between the host, ledger, tasks and threads are traversed in support of many kernel calls. The *callout*, *callout_ptr*, *callout_data*, *callout_object* and *callout_table* template classes encapsulate the mechanisms to traverse these lists, invoking specified methods on the list elements.

Kernel Object Invocation

A message sent to a kernel object will eventually reach the virtual method *kernel_object*::**send**, which is implemented in this subsystem. This MIG generated method determines which kernel interface routine the user is invoking (by examining the *msgh_id* field in the message header) and calls the demultiplexing stub for that interface. This MIG generated stub is responsible for unmarshalling and validating the arguments in the message and for converting any additional kernel rights in the message to locked pointers to their appropriate kernel objects. The MIG stub then calls the target routine, marshals any output arguments into a reply message, and invokes *kernel_object*::**process_reply**, indicating what reply message processing should be performed.

Miscellaneous Classes

The *sample_object* class encapsulates the mechanisms to control PC sampling (for tasks or threads) and to recycle sampling messages.

The *exception* class encapsulates the mechanics to maintain and chose an exception destination.

A message_holder object is an execution point that can be holding (have cached) a message (cached_message) for fast message allocation.

Concurrency Issues

In general, the subsystem's objects are protected by their kernel object lock. When a code path must hold two object locks of different types, the locks must be taken according to the following hierarchy:

- *host*
- *ledger*
- *processor_set*
- *task*
- *thread*

In addition to these object locks, the subsystem employs a set of locks to synchronize the management of object relationships during the deallocation of objects by the harvest_thread located within the *harvest_parent* class. Each of these locks protects the object's queue of child objects.

These locks permit the kernel to synchronize access to the respective object queues during object deallocation without violating the subsystem's locking protocol. For example, when the harvest_thread invokes a thread's **harvest** method to deallocate the thread, it holds the thread lock. Before it deallocates the thread's data structure, the **harvest** method removes the thread from the task's queue of threads. Because it already holds the thread's lock, the method cannot take the task's lock without violating the locking protocol. Instead, the method takes the task's harvest_parent lock to remove the thread from the queue.

Similarly, the destruction of a task requires unlinking the task from its owning ledger, which is done under the protection of the ledger's harvest_parent lock.

Destruction of a task involves destruction of its contained threads (and aborting the accesses of any guest threads). When the last child thread is harvested, that child notices that the parent is to be terminated as well and arranges for the task to be harvested.

Ledger destruction proceeds similarly.

Subsystem Initialization

Initialization of this subsystem consists of the construction of the host and processor set objects, the creation of the harvest thread and the establishment of the dead letter office handler.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 5 User Port Name Space Subsystem Design

Overview

Threads within a task may access a given port only if the task holds a right to that port. A task refers to such a right by a *user port name*. Every task includes a *port name space* that maps the task's user port names to their underlying capabilities. The User Port Name Space subsystem manages this name space.

All kernel mechanisms that relate to a particular task's holding of a capability, including the notion of a particular task's holding of the receive right for a user port, are managed within this subsystem.

The dominant operations performed within this subsystem are the *copying in* of a right — translating a user port name into a capability, moving the right (or deriving another right) out of the user visible space for kernel processing — and the *copying out* of a right — the insertion of a right into the name space, logically moving it from kernel control into user visible space. Both of these operations have numerous variants corresponding to the options presented at the kernel interface for the sending and receiving of rights.

Although the grouping of user ports into port sets is implemented by Connection Management, the ability to name and use a port set as the target of a message receive operation is handled here. It is in User Port Name Space that receive rights and port sets gain their uniformity — this subsystem exports a method that, in either case, returns a transfer queue for the **mach_msg_trap** to use to receive a message from the queue.

There are also methods that implement the user-visible port manipulation interfaces. These include a method that returns the named port's right type, a method that returns the named port's user reference count, a method that modifies the named port's user reference count, a method that moves the named port into or out of a port set, and a method that requests a dead name notification.

Subsystem Interface

The User Port Name Space subsystem exports a single class, the *port_name_space* class. The class exports the following member functions:

- *port_name_space*::**port_name_space** — Initialize the space, billing the specified *space_source*.
- *port_name_space*::**~port_name_space** — Tear down the space, destroying any port rights and port sets remaining.
- *port_name_space*::**copyin** — Extract the named right from the space, returning it as a *message_right*. The supplied *mach_msg_type_name_t* determines the way in which the right should be extracted (copy a send right, move a send right, make a send-once right, etc.).
- *port_name_space*::**copyin_reply** — Optimized reply right **copyin**. Extract the named right from the space via the specified extraction type. Upon failure, a null right is used.
- *port_name_space*::**copyin_request** — Optimized request right **copyin**. Extract the named right from the space via the specified extraction type. Upon failure, the dead right is used.
- *port_name_space*::**copyout** — Insert a *message_right* into the space, returning its assigned name. An overloaded variant additionally returns the *mach_port_type_t* of the right for the convenience of the caller.
- *port_name_space*::**create_dname** — Create a new dead-name right and insert it into the space. The Connection Management subsystem is called to create the new object.
- *port_name_space*::**create_pset** — Create a port set and insert it into the space. The Connection Management subsystem is called to create the new object.
- *port_name_space*::**create_user_port** — Create a user port and insert its receive right into the space. The Connection Management subsystem is called to create the new object.
- *port_name_space*::**destroy** — Destroy the named right as specified by the Kernel Interface **mach_port_destroy**. If necessary, the Connection Management subsystem is called to destroy the underlying object (port set or receive right) and send notifications.
- *port_name_space*::**entry_size** — Determines the resources need to grow the space by an entry.
- *port_name_space*::**get_attributes** — Return the specified attributes of the named port.
- *port_name_space*::**get_refs** — Get the reference count for the named right as specified by the Kernel Interface **mach_port_get_refs**.
- *port_name_space*::**get_type** — Return the type of the named right as specified by the Kernel Interface **mach_port_get_type**.

- *port_name_space*::**mod_refs** — Modify the reference count for the named right as specified by the Kernel Interface **mach_port_mod_refs**.
- *port_name_space*::**move_member** — Perform the action specified by the kernel interface **mach_port_move_member**.
- *port_name_space*::**receive** — Locate and receive from the transfer_queue target for the specified port name.
- *port_name_space*::**request_notif** — Perform the action specified by the kernel interface **mach_port_request_notification**.
- *port_name_space*::**set_attributes** — Set the attributes of the named port as specified.

The *port_name_space* class derives from no other class.

The *user_port_name_space* class (with only a constructor for initialization) represents the subsystem as a whole.

Security Policy

This subsystem implements the per-task port name to right translation, thereby providing a task with a set of rights while restricting it to using only those rights it has been granted.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Connection Management for right to port translation.
- Resident Memory Management for memory allocation.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (port (rights)).
- Provides system facilities used to represent objects and subjects (port (rights)).
- Provides system resources which are used to implement objects and subjects (port (rights)).
- Is expected by other TCB components to function correctly.

Functional Decomposition

The *port_name_space* class is supported by two private classes, *port_name_space_entry* and *table<port_name_space>*. A given port_name_space object contains a port_name_space_entry object for each user port name that has been inserted in the name space. Each port_name_space object has a single table<port_name_space> object (hash table) that is used to implement the coalescence of send and receive rights.

The *port_name_space_entry* class holds an individual entry in the space. This entry contains the information about a right, including its type, its reference count (if it is a send right) and the capability or port_set to which it refers.

The *table<port_name_space>* class maintains a reverse lookup table that maps capabilities to user port names. The subsystem uses this table to coalesce send rights with their respective receive rights. The table maintains information only for port_name_space_entry objects containing send or receive rights; no information is maintained for port sets or send-once rights.

Table 3 shows the object instance relationships for this subsystem.

Table 3 User Port Name Space Object Relationships

Object	References
port_name_space	port_name_space_entry table<port_name_space>

Concurrency Issues

The *port_name_space* class contains no locks. Rather, it is protected by a caller lock (the task) which must be held across any operation on the space. The *port_name_space_entry* and *table<port_name_space>* classes are protected by their containing *port_name_space*.

Subsystem Initialization

This subsystem requires no initialization.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 6 User Address Space Subsystem Design

Overview

The User Address Space subsystem implements task address spaces and manages the task-local view of virtual memory. The subsystem's management consists of tracking each address space's *regions* and enforcing each task's access modes to its own memory.

An address space consists of an ordered (by address) set of *address regions*. There are four types of address regions:

- Named regions.
- Anonymous regions.
- Page list regions.
- Reserved regions.

The first three types represent actual, accessible regions of the address space. Both named and anonymous regions are pageable regions that map data that is managed by the Memory Extent Management subsystem. Data in these region types may be wired (made non-pageable). Page list regions are sets of resident pages not managed by any other subsystem; they support efficient out-of-line transfer of pages for device and external memory management operations. A distinct region is a virtually contiguous span of the address space with the same memory properties. User address space manipulations can span kernel maintained regions — the regions are split as necessary to align with user requests such that each region has a single set of attributes.

Reserved regions specify areas of the address space that are protected against arbitrary (non-explicit) allocation. The property of being reserved is an orthogonal property of a portion of the address space relative to its current method of use. As such, the maintenance of reserved regions is separate from the maintenance of the allocated regions; only when an allocation is to be performed are both lists taken into account.

Most of the operations supported by the User Address Space subsystem correspond to the kernel-exported address space functions. The subsystem also exports methods that support the sending and receiving of out-of-line memory areas by **mach_msg_trap**.

Since this is the subsystem that understands the meaning of a particular task's user address, accesses to the user address space must be made at this subsystem or above. Any faults taken by such accesses are handled by this subsystem; this subsystem is the target for page and other memory faults taken in kernel mode.

Subsystem Interface

Aside from its duties to support the address space of tasks, the User Address Space subsystem is also the target for page and other memory faults taken in kernel mode:

- **check_address_and_size** — Utility routine to validate user supplied (aligned) region parameters.
- **copyin** — Copy a specified range of the calling task's address space into kernel memory.
- **copyin_page_list** — Copy a specified range of the calling task's address space into a set of resident pages. This is used for the copy in of an out-of-line area that cannot be stolen.
- **copyout** — Copy a specified range of kernel memory out to the calling task's address space.
- **copyout_page_list** — Copy the contents of a page list to a range of the user's address space.
- **copyaround** — Copy a range of a user's address space to another range in the user's address space.
- **host_set_syscall_return** — Special purpose routine to write the return value from a system call to the user stack.
- **host_syscall** — Special purpose routine to copyin the argument list and values for a system call.
- **kernel_trap** — Vectored to by the Processor Scheduling subsystem for memory access faults in kernel mode. All memory accesses in kernel mode are made in one of the above listed routines. This fault routine simply calls the **vm_fault** method for the current (or guest) task's address space. This routine differs from the **user_trap** function (in the Kernel Interface subsystem) in that failure returns from this call (e.g., invalid or inaccessible memory) cause the appropriate user memory access routine to return an error, rather than being translated into a user mode exception.
- **touch_memory** — Simulate a reference to a range of user memory. This is done to force copy or wiring mechanisms on the set of pages to be invoked.
- **unaligned_check_address_and_size** — Utility routine to validate user supplied (unaligned) region parameters.

These interfaces are invoked upon an address space object:

- *address_space*::**address_space** — Constructor. Creates an empty address space.
- *address_space*::**~address_space** — Destructor. Destroys all address regions in the space.
- *address_space*::**deliver_pages** — Insert a range of kernel memory into the space.
- *address_space*::**do_copy_on_reference** — Traverse the address space, forcing copy where necessary.
- *address_space*::**inherit** — Inherit regions from the specified address space, copying or sharing regions as appropriate.
- *address_space*::**inherit_structure_size** — Determine the size of structures that are needed to represent an inheritance of this address space.
- *address_space*::**ool_size** — Determine the size of the structures that will be necessary to hold an ool region.
- *address_space*::**stealable** — Determine if the given range holds a page_list.
- *address_space*::**total_virtual_allocation** — Return the total amount of virtual memory allocated in this address space.
- *address_space*::**vm_allocate** — Create a new anonymous region.
- *address_space*::**vm_behavior_set** — Pass the desired behavior down to the underlying memory extents.
- *address_space*::**vm_deallocate** — Remove the specified span of memory from the space.
- *address_space*::**vm_fault** — Determine the reason for the memory fault; if access permitted, call memory object management to make the page resident or accessible.
- *address_space*::**vm_inherit** — Mark the inheritance of the specified span.
- *address_space*::**vm_map** — Map a specified memory object representative into the address space.
- *address_space*::**vm_protect** — Mark the allowed permissions for the span, calling down to change permissions on resident pages accordingly.
- *address_space*::**vm_region** — Return desired memory region information.
- *address_space*::**vm_reserve** — A reserved region is created to mark the section of the address space as reserved.
- *address_space*::**vm_unreserve** — Mark a span of the address space as not reserved, freeing reserved regions as necessary.
- *address_space*::**vm_unwire** — Clear the wired flag for the span.
- *address_space*::**vm_wire** -— Mark the specified span as wired. The caller must ensure that the invoking thread is a guest of the space so that this method can touch the pages to actually make them resident.

The *message_memory* class describes out-of-line memory; it exports the following functions:

- *message_memory*::**copyin** — Copy data in when page stealing was not possible.
- *message_memory*::**copyout** — Copy data out, such as for a **vm_write** call.

- *message_memory*::**deliver** — Insert the memory into the specified address space; the descriptor will no longer contain a page_list.
- *message_memory*::**steal** — Initialize the descriptor to describe a page_list stolen from a specified address region.

The *user_address_space* class (with only a constructor for initialization) represents the subsystem as a whole.

Table 4 shows the inheritance relationship for this subsystem's objects.

Table 4 User Address Space Class Inheritance

Class	Inherits from
address_space	address_translation_map (VM res) mutex (Sched)
allocated_region	address_region
allocated_region_queue	address_region_queue
anonymous_address_region	allocated_region anonymous_virtual_extent (VM ext)
named_address_region	allocated_region named_virtual_extent (VM ext)
page_list_address_region	allocated_region dynamic_space (Space acct)
reserved_address_region	address_region
reserved_region_queue	address_region_queue
user_address_space	exception_vector (Sched)

Security Policy

This subsystem is responsible, at the top level, for enforcing access to memory. For the most part, this consists of checking the task's allowed access modes whenever an attempt is made to access memory (actually when an attempt is made to access a page not known to be validly accessible, namely at page fault time). This subsystem must also correctly pass the (passed in) task security ID to memory extent management for the map operation for validating use of the memory object representative.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Memory Extent Management.
- Resident Memory Management for memory allocation.

- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Enforces access mediation decisions on memory objects.
- Provides system facilities used to represent objects and subjects (memory).
- Provides system resources which are used to implement objects and subjects (memory).
- Supports architectural requirements for trust (process isolation).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 5 shows the object instance relationships for this subsystem.

Table 5 User Address Space Object Relationships

Object	References
address_region_queue	address_region
address_space	allocated_region_queue reserved_region_queue space_source (Space acct)
page_list_address_region	page_list (VM res)

The User Address Space subsystem only exposes the *address_space* to its callers. This object derives from the *address_translation_map* that maps the space.

An address_space contains an ordered set of address_region objects (held in an address_region_queue, either allocated_region_queue or reserved_region_queue). The four different types of regions represent four different derived classes:

- *named_address_region* derives from *allocated_region* and from *named_virtual_extent*.
- *anonymous_address_region* derives from *allocated_region* and from *anonymous_virtual_extent*.
- *page_list_address_region* derives from *allocated_region* and *dynamic_space*.
- *reserved_address_region* derives simply from *address_region*.

As the address space is manipulated, these address region objects are split accordingly and modified individually.

Concurrency Issues

An address_space object and child address_region objects are protected by a single lock. Because **vm_fault** is the dominant operation, and this operation does not modify the address space, the subsystem may benefit from having a single-writer/multiple-reader lock.

Subsystem Initialization

This subsystem requires no initialization.

Languages Employed

Except for machine-dependent assembly language logic in **copyin**, **copyout** and related routines, this subsystem is implemented in C++.

CHAPTER 7 Device Management Subsystem Design

Overview

The Device Management subsystem implements the *kernel device* abstraction, which provides a simple interface to the kernel device drivers in the Clock and Device Services subsystem. Most of the actual mechanisms supporting kernel devices lie in the Clock and Device Services subsystem. Among other reasons, the Device Management subsystem exists above Connection Management in order to provide first-class kernel objects representing the devices.

This subsystem exports and manages the character string names for kernel devices, which are used as input to the **device_open** call. The open and close support is a major part of Device Management.

Since the Device Management subsystem provides the multiple device objects/ports that permit multiple access paths to any shared device, this subsystem enforces access control for device operations.

In addition to the functions to open and close kernel devices, the subsystem also exports functions to read and write device data, get and set device state and to perform device-specific special operations corresponding to the exported kernel interfaces.

Residing above both the Clock and Device Services and the Memory Extent Management subsystems, Device Management provides the mechanisms to create and manage memory objects representing shared memory versions of the devices.

Subsystem Interface

The services provided by the Device Management subsystem exactly correspond to the device services described in the *Kernel Interfaces* document.

Device State Services

The state of open kernel_devices may be interrogated and modified by the owning task. All requests are handled synchronously.

- **convert_right_to_device** — MIG kernel-stub called routine to validate a kernel device right.
- *kernel_device*::**~kernel_device** — Destructor; ensures that kernel_device is closed.
- *kernel_device*::**device_close** — Terminate all activity associated with the logical device and destroy the kernel_device object.
- *kernel_device*::**device_get_status** — Return the current device status.
- *kernel_device*::**device_map** — Establish a memory object representing the device.
- *kernel_device*::**device_set_filter** — Establish selection criteria by which incoming network packets can be routed to a specific port.
- *kernel_device*::**device_set_status** — Set device status.
- *kernel_device*::**resource_destroy** — Indicate that the resource named by this kernel object should be destroyed.

Data Transfer Services

Both inband (using data contained in the request/response messages) and out-of-band (using pointers in the request/response messages to separately-allocated buffers) are supported. All data transfer requests to the Clock and Device Services subsystem are asynchronous.

- *kernel_device*::**device_read** — Read a sequence of bytes from the kernel device.
- *kernel_device*::**device_read_inband** — Read an “inband” sequence of bytes.
- *kernel_device*::**device_write** — Write a sequence of bytes to a kernel device.
- *kernel_device*::**device_write_inband** — Write an “inband” sequence of bytes.

To keep from exhausting the supply of network filter messages, the messages supplied to the device_filter are instantiations of the *recycling_msg* class.

- *recycling_msg<kernel_device>*::**recycling_msg** — Constructor.
- *recycling_msg<kernel_device>*::**~recycling_msg** — Destructor.
- *recycling_msg<kernel_device>*::**disable_recycling** — Marks the storage so the next attempted deletion will not recycle the message, but deallocate it instead.
- *recycling_msg<kernel_device>*::**operator new** — Allocates the filter packet storage, constructing its allocation record.
- *recycling_msg<kernel_device>*::**operator delete** — If the kernel_device that owns this message exists, reuse the message buffer.

Device Control

The class *device_control* provides a persistent model of the set of user-accessible devices. This class exports the following member function:

- *device_control*::**open_device** — Request that a physical device be opened.

Unlike the other device I/O messages, *device_open_reply* (from Clock and Device services) has a device management component concerned with open failure.

- *device_open_reply*::**~device_open_reply** — Destructor; checks the result of the operation and cleans up after failure of **open**.

The *device_management* class represents the subsystem as a whole.

- *device_management*::**device_management** — Initialize the device tables.
- *device_management*::**lookup_device_name** — Locate the device_control for a given device name.

Table 6 shows the inheritance relationship for this subsystem's objects.

Table 6 Device Management Class Inheritance

Class	Inherits from
device_control	dynamic_space (Space acct) mutex (Sched)
device_management	mutex (Sched)
kernel_device	kernel_object (IPC conn)

Security Policy

This subsystem is responsible for ensuring that all kernel_device operations originate from a task whose security ID matches that for which the kernel_device object was created. (Note that each distinct **device_open** call for the same physical device results in a distinct kernel_device object representing that opening.) It also assures that all requested data transfer operations are compatible with the opening modes for the kernel_device.

Privileges and Capabilities

This subsystem has access to the device_control object and the ability to establish new kernel_device objects and ports with specified access modes in order to support **device_open**.

Use of Protection Mechanisms

This subsystem uses the security ID tag from messages passed to it as a means of enforcing access control for devices.

This subsystem requires the services of the following subsystems:

- Memory Extent Management to establish device memory objects.
- Connection Management for right to port translation.
- Clock and Device Services for the physical devices themselves.
- Resident Memory Management for memory allocation.
- Processor Scheduling for synchronization.

Protection Criticality

This subsystem is protection critical because it:

- Enforces access mediation decisions on devices.
- Provides system facilities used to represent objects and subjects (devices).
- Provides system resources that are used to implement objects and subjects (devices).
- Maintains separation of disk partitions.
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 7 shows the object instance relationships for this subsystem.

Table 7 Device Management Object Relationships

Object	References
device_control	hw_device (Dev svc) page_list (VM res) event (Sched)
device_management	devtab_entry
kernel_device	device_control hw_device (Dev svc) device_filter (Dev svc)

Kernel devices are implemented using the *kernel_device* object class, which derives from the *kernel_object*. The corresponding device_control object is used as the starting point for the creation of a kernel_device object each time a device_open request is made.

The *device_control* class models the system I/O device configuration. At subsystem initialization, *device_management*::**device_management** scans the configuration, creating the corresponding device_control objects used by the **device_open** function. Each device_control object has a primary name, derived from its type and unit number, which is registered in the device name table. Additional “console” names and the “boot_device” are assigned as aliases in the same table.

Device services are implemented as methods on the kernel_device. The kernel_device object includes a pointer to the hw_device to which access has been granted. The requestor's security ID is checked against that of the kernel_device's creator and the type of request must be consistent with the opening modes.

A device that can be mapped maintains a memory cache object associated with the device_control object. Establishment of a mapped device by a call to kernel_device::**device_map** results in the creation of a memory_object_representative, for which a capability is kept in the kernel_device object. Mapping support is device-specific; no generalized virtual I/O support is provided for mapped devices.

A device with an attached input filter has an attached device_filter object, one per kernel_device object.

Device I/O requests require the creation of an io_message_request object, a message to the *response port* that will signal completion. The hw_device's interrupt handler will fill in and send the io_message_request object as the response message. As a *message* subclass, the io_message_request object's memory will thus be freed by the IPC copyout procedure.

Concurrency Issues

The creation of kernel_device objects by *device_control*::**open_device** is done under the control of a mutex lock embedded in the device_control entry. The newly created kernel_device is automatically locked as part of kernel object creation.

The *kernel_device*::**close** operation is also performed under the embedded mutex lock in the corresponding device_control entry.

All other kernel_device operations are performed under the mutex lock embedded in the kernel_object from which it derives. Synchronous requests must relinquish the mutex at the same time that they block waiting for I/O completion.

The **read** operation differs from the other operations in that a buffer must be allocated, possibly a large one. This operation uses the **reserve** memory allocator for this purpose, to avoid stalling the thread (and possibly others) indefinitely waiting for memory utilization to get low enough. If **reserve** enters an interruptible wait, it releases the device mutex (which is passed in). The **read** operation must detect this case and attempt to relock the device object (given its capability).

Subsystem Initialization

Subsystem initialization consists of querying the system configuration, constructing the corresponding device_control objects and initializing them in turn. The device name table is also created.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 8 Clock Management Subsystem Design

Overview

The Clock Management subsystem implements the kernel *clock* and *alarm* abstractions. Kernel clocks allow tasks to synchronously query a representation of the current time. Kernel alarms allow tasks to request asynchronous notification of the passage of time, via messages sent by the kernel to a specified port.

The following types of clock services are provided, of which only the first is privileged.

- Set the time and/or resolution of the clock.
- Obtain the time and/or resolution of the clock.
- Map the clock to a memory object.
- Sleep until a specified time.

The following types of alarm services are provided, none of which is privileged.

- Create a kernel alarm, which handles specific alarm requests.
- Request an alarm at a specified time.
- Request a periodic series of alarms, at a specified interval, starting at a specified time.
- Cancel a kernel alarm's outstanding alarm request (if any).
- Terminate a kernel alarm (canceling any outstanding alarm request).

These services correspond to the kernel interface's exported time services.

Most of the work involved with the kernel's time services is actually performed by the Clock and Device Services subsystem. Unlike the Clock and Device Services subsystem, the Clock Management subsystem exists above Connection Management in order to be able to establish first-class kernel objects representing kernel clocks and alarms, which directly correspond to their physical counterparts.

Subsystem Interface

The services provided by the Clock Management subsystem correspond to the clock services described in the *Kernel Interfaces* document. Instances of the *kernel_clock* objects are created for the following clock types: *realtime_clock*, *battery_clock*, and *highres_clock*. Any of these clock types that are not physically configured will be identified as such at system initialization (except the *realtime_clock*, which must always exist).

System Initialization & Shutdown

The Clock Management subsystem is itself represented by a static clock_management object, which provides standard class methods to support orderly initialization and shutdown of the kernel.

- *clock_management*::**clock_management** — This static member function queries the system configuration, constructs the corresponding kernel_clock objects, and initializes them in turn.
- *clock_management*::**get_clock** — This static member function returns a clock of the specified type.

Kernel Clock Services

Kernel clock services depend upon access to the respective kernel_clock object, of which they are implemented as class functions.

- **convert_control_right_to_clock** — MIG kernel-stub called routine to validate a clock control right.
- **convert_name_right_to_clock** — MIG kernel-stub called routine to validate a clock name right.
- *kernel_clock*::**kernel_clock** — Constructor.
- *kernel_clock*::**~kernel_clock** — Destructor.
- *kernel_clock*::**alarm_create** — Create a kernel_alarm object and request that the clock send a *alarm_reply* message to the specified port at a future time. This implements the kernel interface **alarm_create** and is called from a ledger.
- *kernel_clock*::**clock_get_attributes** — Return the current resolution of the clock and other operational attributes.
- *kernel_clock*::**clock_get_time** — Return the current time kept by the clock. Except when the protected **clock_set_time** function is performed, clock time is guaranteed to be monotonically increasing.
- *kernel_clock*::**clock_map_time** — Return a memory object representative port that represents a read-only memory object into which the clock time is mapped at relative location zero.
- *kernel_clock*::**clock_set_attributes** — Set the current resolution and operational attributes of the clock.

- *kernel_clock*::**clock_set_time** — Set the time kept by the clock. Any pending alarms will be terminated with time-stamps set to the former clock value.
- *kernel_clock*::**clock_sleep** — Cause the requesting thread to be suspended until the specified future time.
- *kernel_clock*::**resource_destroy** — Virtual destroy method, simply deletes the object.

Alarm Services

Alarm services use the respective kernel_clock object and create/use a kernel_alarm object, which is derived from both the *alarm_message_request* and *kernel_object* classes.

- **convert_right_to_alarm** — MIG kernel-stub called routine to validate an alarm right.
- *kernel_alarm*::**kernel_alarm** — Constructor.
- *kernel_alarm*::**~kernel_alarm** — Destructor.
- *kernel_alarm*::**alarm_cancel** — Cancel a scheduled alarm.
- *kernel_alarm*::**alarm_set** — Set up the alarm_message_request and set the alarm on the corresponding hw_clock.
- *kernel_alarm*::**alarm_terminate** — Cancel any scheduled alarm and destroy the kernel_alarm object.
- *kernel_alarm*::**resource_destroy** — Cause the resource named by this kernel object to be destroyed

Alarm Message Services

The alarm message services instantiate the *recycling_msg* class to provide the alarm messages that recycle against the kernel alarm object.

- *recycling_msg<kernel_alarm>*::**recycling_msg** — Constructor.
- *recycling_msg<kernel_alarm>*::**~recycling_msg** — Destructor.
- *recycling_msg<kernel_alarm>*::**disable_recycling** — Marks the storage so the next attempted deletion will not recycle the message, but deallocate it instead.
- *recycling_msg<kernel_alarm>*::**operator new** — Allocates the alarm message storage, constructing its allocation record.
- *recycling_msg<kernel_alarm>*::**operator delete** — If the kernel_alarm that owns this message exists, reuse the message buffer.

Table 8 shows the inheritance relationship for this subsystem's objects.

Table 8 Clock Management Class Inheritance

Class	Inherits from
kernel_alarm	kernel_object (IPC conn) alarm_message_request (Clk svc)

Table 8 Clock Management Class Inheritance

Class	Inherits from
kernel_clock	kernel_object (IPC conn)

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Connection Management for right to port translation.
- Clock and Device Services for clock support.
- Resident Memory Management for memory allocation.
- Transfer Management for alarm messages.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 9 shows the object instance relationships for this subsystem.

Table 9 Clock Management Object Relationships

Object	References
clock_management	kernel_clock
kernel_clock	hw_clock (Clk svc)

All clock services are implemented as methods on the *kernel_clock* class. The *kernel_clock* is derived from the *kernel_object* class (in Connection Management) and contains a reference to its hw_clock object (in Clock and Device Services). Instances are created for the *realtime*, *battery*, and *highres* clocks.

Alarm services are implemented as class functions on the *kernel_alarm* class, which is derived from the *alarm_message_request* class in the Clock and Device Services subsystem and from the *kernel_object* class. The kernel_alarm object supports cancellation and rescheduling of the alarm_request from which it derives.

Concurrency Issues

The kernel_clock object holds a mutex in its kernel_object, which is locked prior to each function call and freed before returning to the caller. With one exception, all services are immediate; the respective locks are held for the duration of the operation. **clock_sleep** must release the lock when it makes the thread wait for its wakeup event.

The kernel_alarm object's operations are controlled by an embedded mutex lock, which is held during the execution of each of its callable class functions.

It is the responsibility of the caller to ensure sequential access to alarm_message objects.

Subsystem Initialization

Subsystem initialization consists of querying the system configuration, constructing the corresponding kernel_clock objects, and initializing them in turn.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 9

Memory Extent Management Subsystem Design

Overview

The Memory Extent Management subsystem provides the system-wide view of virtual memory. It is the central component in the implementation of the kernel's memory management services because it coordinates the interactions between the other subsystems and entities that participate in memory management, namely the User Address Space Management subsystem, the Resident Memory Management subsystem, and the system's memory managers.

The subsystem provides the following services:

- Initialization and destruction of memory objects.
- The tracking of multiple mappings to a given memory object.
- Implementation of memory sharing and virtual copy operations.

The most visible part of this management is the naming of external memory. This subsystem implements memory cache objects; these objects are used by the memory managers to perform paging operations and cache management operations. This subsystem implements the memory object initialization and termination protocols. It tracks all known memory object representative ports and abstract memory object ports and their relationship to one another and to their memory cache objects. In this way the subsystem manages the access rights permitted to particular security IDs, rejecting attempts to map inaccessible objects and reporting the permitted access modes to the User Address Space subsystem, which enforces these permissions for objects that are accessible.

The management of the relationship of memory to other memory consists of the tracking of memory sharing and virtual copying. Toward this end, this subsystem exports objects that represent active (mapped somewhere) extents of virtual memory — named_virtual_extent and anonymous_virtual_extent objects. It is quite possible for two different virtual extents to reference overlapping sets of pages.

The kernel tracks memory relationships so that it can determine when a given unit of memory may be terminated or deleted. For anonymous memory, when no virtual extent references a given page, the page can be discarded; for named memory, when no virtual extent references any of the pages of the associated memory object (or a not-yet copied virtual copy of one) the memory object can be terminated.

Residing above Connection Management allows this subsystem to provide the user visible memory cache object, as well as to establish the virtual paging object provided for the default memory manager. Also, this subsystem exports services so that higher layer kernel subsystems can create memory objects representing kernel entities to export to user tasks. Such entities include mapped devices and clocks.

Subsystem Interface

The Memory Extent Management subsystem exports the following object classes:

- *named_virtual_extent* — Represents a specific mapping to a memory object containing named data.
- *anonymous_virtual_extent* — Represents a specific mapping to a set of virtual pages containing anonymous data.
- *virtual_extent* — Basis for both *named_virtual_extent* and *anonymous_virtual_extent*.
- *memory_cache* — Represents the set of a memory object's pages that are currently cached in resident memory, as seen by the memory manager; the memory manager sends cache control messages to the kernel via this object.
- *virtual_paging* — Represents the area of secondary storage the kernel uses to page-out anonymous data.
- *vm_map_state* — Used to encapsulate the interaction state between memory extent management and its callers during the initialization of named memory objects.

In addition, the *memory_object_rep* class, although its instances are not exported, the class does export some static methods.

The *memory_extent_management* class (with only a constructor for initialization) represents the subsystem as a whole.

named_virtual_extent Object Interface

The *named_virtual_extent* class exports the following methods:

- *named_virtual_extent*::**named_virtual_extent** (vm_map_state) — This constructor performs the final stage of the memory object initialization sequence. It binds the new extent to the memory_object_rep and named_virtual_memory objects specified by the vm_map_state.

- *named_virtual_extent*::**named_virtual_extent** (named_virtual_extent, *split*) — Split the given extent object into two objects at a specified boundary point by shortening the original object to represent the beginning of the extent and constructing a new object to represent the end of the extent.
- *named_virtual_extent*::**named_virtual_extent** (named_virtual_extent, *share*) — Construct a new virtual extent object that duplicates (shares memory with) the original. This duplicate constitutes an additional reference to the pages so described.
- *named_virtual_extent*::**~named_virtual_extent** — Destructor.
- *named_virtual_extent*::**unmap** — Remove any pages specified by this object from the specified address_translation_map.
- *named_virtual_extent*::**unwire** — Unwire the specified extent.
- *named_virtual_extent*::**vm_fault** — Handle a page fault.
- *named_virtual_extent*::**wire** —Wire the specified extent.

anonymous_virtual_extent Object Interface

The *anonymous_virtual_extent* class exports the following methods:

- *anonymous_virtual_extent*::**anonymous_virtual_extent** — Construct a new anonymous virtual memory extent of a given size. None of the pages yet exist.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (page_list) — Construct a new anonymous virtual memory extent to represent the list of resident pages provided in the call. An argument optionally allows the pages to be marked as wired.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (anonymous_virtual_extent, *split*) — Split the given extent object into two objects at a specified boundary point by shortening the original object to represent the beginning of the extent and constructing a new object to represent the end of the extent.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (anonymous_virtual_extent, *copy*) — Construct a copy representation of the specified extent. The result is an anonymous virtual extent whose pages have one map reference. If the underlying object has a copy strategy of force copy, the virtual copy will be marked as copy-on-reference; the caller must touch the pages to force the copy.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (named_virtual_extent, *copy*) — Construct a copy representation of the specified extent. The result is an anonymous virtual extent whose pages have one map reference. If the underlying object has a copy strategy of force copy, the virtual copy will be marked as copy-on-reference; the caller must touch the pages to force the copy.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (anonymous_virtual_extent, *share*) — Construct a new virtual extent object that duplicates (shares memory with) the original. This duplicate constitutes an additional reference to the pages so described.
- *anonymous_virtual_extent*::**anonymous_virtual_extent** (vm_map_state) — This constructor is similar to *named_virtual_extent*::**named_virtual_extent** (vm_map_state), except the result is an anonymous virtual extent representing a virtual copy of the named memory object. If the underlying object has a copy strategy of

force copy, the virtual copy will be marked as copy-on-reference; the caller must force the copy by touching the pages.

- *anonymous_virtual_extent*::**~anonymous_virtual_extent** — Destructor.
- *anonymous_virtual_extent*::**unmap** — Remove any pages specified by this object from the specified address_translation_map.
- *anonymous_virtual_extent*::**unwire** — Unwire the extent.
- *anonymous_virtual_extent*::**vm_fault** — Handle a page fault.
- *anonymous_virtual_extent*::**wire** —Wire the specified extent.

Common virtual_extent Object Interfaces

Most *virtual_extent* methods are embedded in the derived classes, or are virtual. The following few interfaces are directly exported.

- *virtual_extent*::**get_offset** — Return the offset of the start of the extent with respect to its underlying memory object.
- *virtual_extent*::**get_representative_port** — Return the capability through which this extent was mapped.
- *virtual_extent*::**get_size** — Return the size of the extent.
- *virtual_extent*::**is_wired** — Determine whether the extent is wired.

memory_cache Object Interface

The *memory_cache* class exports the following methods in support of (external) memory cache control calls:

- **convert_right_to_memory_cache** — MIG kernel-stub called routine to validate a memory cache right.
- *memory_cache*::**memory_cache** — Constructor.
- *memory_cache*::~**memory_cache** — Destructor.
- *memory_cache*::**create_kernel_object** — Given a set of resident pages, this static member function creates a memory cache object and associated named virtual memory object with the pre-existing set of pages, all of which will be marked as wired. The copy strategy for this object is set to none to force any copies attempted. The existence of this object qualifies as a reference to the newly created “named” object, so the object will be protected from termination even though it may not ever be mapped into any address space.
- *memory_cache*::**get_statistics** — Return **host_statistics** object cache statistics.
- *memory_cache*::**memory_object_change_attributes** — Set the attributes (copy strategy, cluster size, anonymous). In general this has no immediate effect; however, clearing of cacheability attribute will terminate the object if the object is currently unmapped.
- *memory_cache*::**memory_object_data_error** — Indicate that the memory manager cannot provide the kernel with the data for the specified region.
- *memory_cache*::**memory_object_data_supply** — Provide the kernel with a specified range of data for a memory object.

- *memory_cache*::**memory_object_data_unavailable** — Indicate that the data requested for the given region should just be zero filled.
- *memory_cache*::**memory_object_destroy** — Destroy the memory cache object structure associated with the named virtual memory object. This operation does not destroy the associated named virtual memory object; that object is destroyed when it becomes completely unmapped.
- *memory_cache*::**memory_object_get_attributes** — Return object attributes.
- *memory_cache*::**memory_object_lock_request** — Restrict access as specified to memory object data.
- *memory_cache*::**memory_object_revoke** — Mark the associated representative structure as dead and force all tasks to recheck their access.
- *memory_cache*::**resource_destroy** — Indicate that the resource named by this kernel object should be destroyed.

virtual_paging Object Interface

The kernel creates and manages a single instance of this class. Most operations performed on this object are implemented within the Resident Memory Management subsystem; the lone exception is implemented within the Memory Extent Management subsystem:

- **convert_right_to_virtual_paging** — MIG kernel-stub called routine to validate a virtual paging object right.
- *virtual_paging*::**virtual_paging** — Constructor; locked on return.
- *vvirtual_paging*::**memory_object_data_error** — Respond to a previous request indicating a data error.
- *virtual_paging*::**memory_object_data_supply** — Respond to a previous request for specified data.
- *virtual_paging*::**resource_destroy** — Indicate that the resource named by this kernel object should be destroyed.
- *irtual_paging*::**vm_set_default_memory_manager** — Supply the kernel with the abstract backing space port (connection_agent).
- *virtual_paging*::**vm_set_recycle_limit** — Set the highest page-in space address to be used.

memory_object_rep Object Interface

The class *memory_object_rep* exports the following static member functions:

- *memory_object_rep*::**create_kernel_representative** — Create a new kernel object representative that names an existing kernel memory object.
- *memory_object_rep*::**destroy_kernel_representative** — Destroy a kernel object representative that names a kernel memory object, revoking access appropriately.
- *memory_object_rep*::**map** — Map any IO ports associated with the representative.
- *memory_object_rep*::**memory_object_establish** — This method associates a memory object representative with the new memory_cache object supplied to the memory manager in the **memory_object_notify** message. It completes the initial-

ization of the memory_cache object and wakes up any threads waiting on the representative.

- *memory_object_rep*::**memory_object_representative** — This method associates a memory object representative with a previously initialized memory_cache object, waking up any threads waiting on the representative object.
- *memory_object_rep*::**set_host_name_port** — Record the host name capability to send in memory_object_notify messages.
- *memory_object_rep*::**unmap** — Remove mappings for any IO ports associated with the representative.
- *memory_object_rep*::**vm_map** — This method maps a named memory object. If the object has already been initialized, the method returns immediately with the vm_map_state needed by the caller to create a virtual extent (or with an error if this representative does not allow the task to map the memory). Otherwise, the method takes one of the following actions:
    - If the object has not been initialized, but the kernel has already allocated a representative structure corresponding to the memory object representative port (the supplied capability), then the underlying memory object is being initialized by another thread. The kernel increments the map reference count contained in the representative structure and arranges for the caller to wait for that operation to complete, releasing all locks in the process.
    - If the kernel has not previously allocated a representative structure, the thread must initiate a full-blown initialization operation on the memory object. The subsystem allocates a new memory_cache object and a new memory_object_rep structure and sends an initialization request (**memory_object_notify**) to the memory manager via the memory object representative port. The caller enters an interruptible wait, releasing all locks in the process (the caller must not hold any locks).

  When the thread wakes up, it returns with an intermediate vm_map_state. The caller is expected to reacquire its lock and invoke the method again with the intermediate state. The second invocation will find the representative initialized.

vm_map_state Object Interface

The class *vm_map_state* exports the following member functions:

- *vm_map_state*::**vm_map_state** — Constructor.
- *vm_map_state*::**~vm_map_state** — Destructor.
- *vm_map_state*::**map** — Map any IO ports associated with the underlying representative.

Table 10 shows the inheritance relationship for this subsystem's objects.

Table 10 Memory Extent Management Class Inheritance

Class	Inherits from
anonymous_memory_node	virtual_memory copy_tree_node anonymous_resident_memory (VM res)
anonymous_virtual_extent	virtual_extent
anonymous_virtual_memory	virtual_memory_extents anonymous_memory_node
anonymous_virtual_memory_root	copy_tree_root anonymous_virtual_memory
copy_tree_root	mutex (Sched)
memory_cache	memory_object
memory_object	kernel_object (IPC conn)
memory_object_rep	event (Sched)
memory_snapshot	anonymous_memory_node
named_virtual_memory	virtual_memory copy_tree_root copy_tree_node virtual_memory_extents named_resident_memory (VM res)
named_virtual_extent	virtual_extent
virtual_paging	memory_object resident_paging (VM res)

Security Policy

This subsystem is responsible for checking the security ID of a task into which a memory object representative is being mapped against the ID associated with the representative.

Privileges and Capabilities

As part of implementing **memory_object_establish** and **memory_object_representative**, this subsystem can establish new memory representatives with a given identity.

This subsystem requires the services of the following subsystems:

- Connection Management for right to port translation.
- Resident Memory Management for memory allocation and page management.

- Transfer Management for enqueing and transfer of messages.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

This subsystem uses the task security ID (passed in) for enforcing access decisions at memory map time.

Protection Criticality

This subsystem is protection critical because it:

- Enforces access mediation decisions on memory objects.
- Implements data structures used in mediation (memory).
- Provides system facilities used to represent objects and subjects (memory).
- Provides system resources which are used to implement objects and subjects (memory).
- Supports architectural requirements for trust (process isolation).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 11 shows the object relationships for this subsystem.

Table 11 Memory Extent Management Object Relationships

Object	References
anonymous_memory_node	anonymous_virtual_memory
anonymous_virtual_extent	anonymous_virtual_memory memory_object_rep
copy_tree_node	anonymous_memory_node anonymous_virtual_memory_root anonymous_virtual_memory copy_tree_root
copy_tree_root	anonymous_memory_node copy_tree_node anonymous_virtual_memory
memory_cache	named_virtual_memory cap_table mutex (Sched)
memory_object_rep	cap_table mutex (Sched)
named_virtual_extent	memory_object_rep named_virtual_memory

Table 11 Memory Extent Management Object Relationships

Object	References
vm_map_state	named_virtual_memory memory_object_rep
virtual_memory_extents	virtual_extent

This subsystem straddles the boundaries between virtual memory as described to Resident Memory Management (*named_resident_memory* and *anonymous_resident_memory*) and virtual memory as described to external memory managers (*memory_cache*) and virtual memory as described to the User Address Space (*named_virtual_extent* and *anonymous_virtual_extent*).

Anonymous Memory

For each new range of anonymous memory created, a new anonymous_virtual_memory object (derived from *anonymous_resident_memory*) is created which lives as long as any virtual extents (virtual_memory_extents) describe parts of it (or are not-yet copied parts of it). The resource billing of the *anonymous_virtual_extent* is up to the caller (it would be billed to the task's address space). When billed, each extent object also bills the associated anonymous_virtual_memory object.

The main purposes of an anonymous_virtual_extent object are to remember that a given set of anonymous pages is referenced somewhere, to remember whether that reference wires them, and, most importantly, to remember the associated anonymous_virtual_memory.

If an anonymous_virtual_extent is a copy of named memory, it will also locate a memory_object_rep structure, which the kernel uses to represent the relationship between a memory object representative port (and any mappings obtained through one) and the target memory object. This memory_object_rep structure is billed with all virtual_extent objects which reference it. Each page fault taken on an extent must recheck its access with the memory_object_rep to ensure the memory manager has not revoked access to the representative (via **memory_object_revoke**).

Named Memory

The structure of objects for named virtual memory derives in a similar way given regard for resource management issues. Virtual memory is considered to belong to address spaces, and is billed there, as opposed to being considered as part of the memory manager (actually, its ledger). As such, the memory_cache kernel object which external memory managers invoke is not derived from the underlying named_virtual_memory object (although the latter does derive from *named_resident_memory*). The memory_cache merely serves to locate the named_virtual_memory object; destruction of the memory_cache object does not destroy the mapping structures in any tasks (although their use of the named object will be severely affected!).

The set of all memory_object_rep objects is kept in a list (cap_table), searched when some task attempts to map a (possibly already known) memory object representative port. Likewise, all memory_cache objects are kept in a list, searched when a memory manager names an abstract memory object port when completing object initialization.

Virtual Copy

One of the important relationships maintained by this subsystem is the virtual copy relationship. The virtual copy mechanism described below handles the copying of both named and anonymous data.

Suppose that task A allows task B to virtually copy a given region of memory. The tasks share copy-on-write access to the data and as long as neither task writes the data, they both access the data from the data's original virtual memory object. The Memory Extent Management subsystem allocates a new anonymous_virtual_memory object to hold pages modified by task B and a new memory_snapshot object to hold copies of pages modified by task A.

If task B attempts to write a specific page of the data, Memory Extent Management invokes the Resident Memory Management subsystem to produce a copy of the page, then inserts the copied page into the new virtual memory object. Task B's modifications are made to the new page within the new virtual memory object and Task A retains access to the unmodified page within the original virtual memory object. Should task B attempt to write data on other pages, the kernel copies each page first and places it in the second virtual memory object.

If task A attempts to write a specific page of the data, the subsystem *pushes* a copy of the page to the memory_snapshot object. When this operation is complete, task A may write the data within the original virtual memory object. Should task B attempt to access the data associated with this page, it will find the data in the memory_snapshot. This mechanism insures that task B retains access to the data as it existed when the virtual copy was made. The first part of Figure 22 shows the results of a virtual copy operation between tasks A and B after both tasks have written the data. The time at which the copy operation took place is *time x.*

Now suppose that task A allows task C to copy the data at *time y*. The second part of Figure 22 shows the development of the virtual copy tree after tasks A and C have written the data. As shown in the figure, the kernel has allocated new objects and inserted them between the original object and the objects allocated for task B's virtual copy of the data.

Now suppose task C allows task D to copy its version of the data at *time z*. The third part of Figure 22 shows the development of the virtual copy tree after tasks C and D have written the data. As shown in the figure, the kernel has allocated new objects and chained them off the virtual memory object that contains C's modifications of the data.

Note that the virtual memory objects that are created as a result of virtual copy operations are anonymous virtual memory objects. Note, also, that as an optimization, the kernel may omit the memory snapshot object between a node and its leaf (tasks B and D in this case). The snapshot is necessary only when the parent object feeds multiple virtual copies.

Concurrency Issues

The most frequently invoked operation supported by Memory Extent Management is **vm_fault**; this routine is called to resolve page faults and initiates page-in of data and access processing. **vm_fault** is invoked far more frequently than any of the other operations supported by this subsystem. By design, **vm_fault** does not modify any structures managed by Memory Extent Management.

A single lock protects each copy tree and is contained in the virtual_memory object at the root of the tree. Since the tree's structures will not be destroyed while any outstanding map references exist, it is possible to locate the top virtual_memory object without yet obtaining its lock; however, the lock must be held before traversing or manipulating the tree. The root virtual_memory object is not destroyed until the last tree element is removed.

virtual_extent objects are assumed to be protected by a caller lock.

The locks protecting the virtual_memory objects protect the existence of their base resident_memory objects as calls are made into Resident Memory Management. *resident_memory*::**vm_fault**, though, just like the *virtual_memory* equivalent, may need to release its caller's lock if it is necessary to enter an interruptible wait for a page. In the fault path, it is necessary for the caller of Memory Extent Management to pass in its lock, which is released after Memory Extent Management obtains its lock. In turn, the Memory Extent Management lock is released by Resident Memory Management after it has obtained its lock.

Much of the activity of Memory Extent Management concerns manipulating data structures in support of virtual copy and the relationship of that to page fault processing. The other significant role of Memory Extent Management is to handle the memory object initialization and termination sequences.

An attempt to map a named object via a memory object representative port for the first time invokes the initialization sequence. The kernel creates a memory_object_rep structure, which is protected by a global lock for all representatives. It also creates a dummy memory_cache object (actually a discard_connection), whose capability is returned in the **memory_object_notify** message; the list of all memory cache objects is protected by a global lock. The calling thread enters an interruptible wait. If the manager responds with **memory_object_establish**, the kernel creates a real memory_cache object named by the same capability, and a named_virtual_memory object associated with the memory_cache. If the manager responds with a **memory_object_representative**, the kernel discards the dummy object and associates the representative with the existing memory_cache object. Either way, the kernel wakes any threads waiting on the representative port.

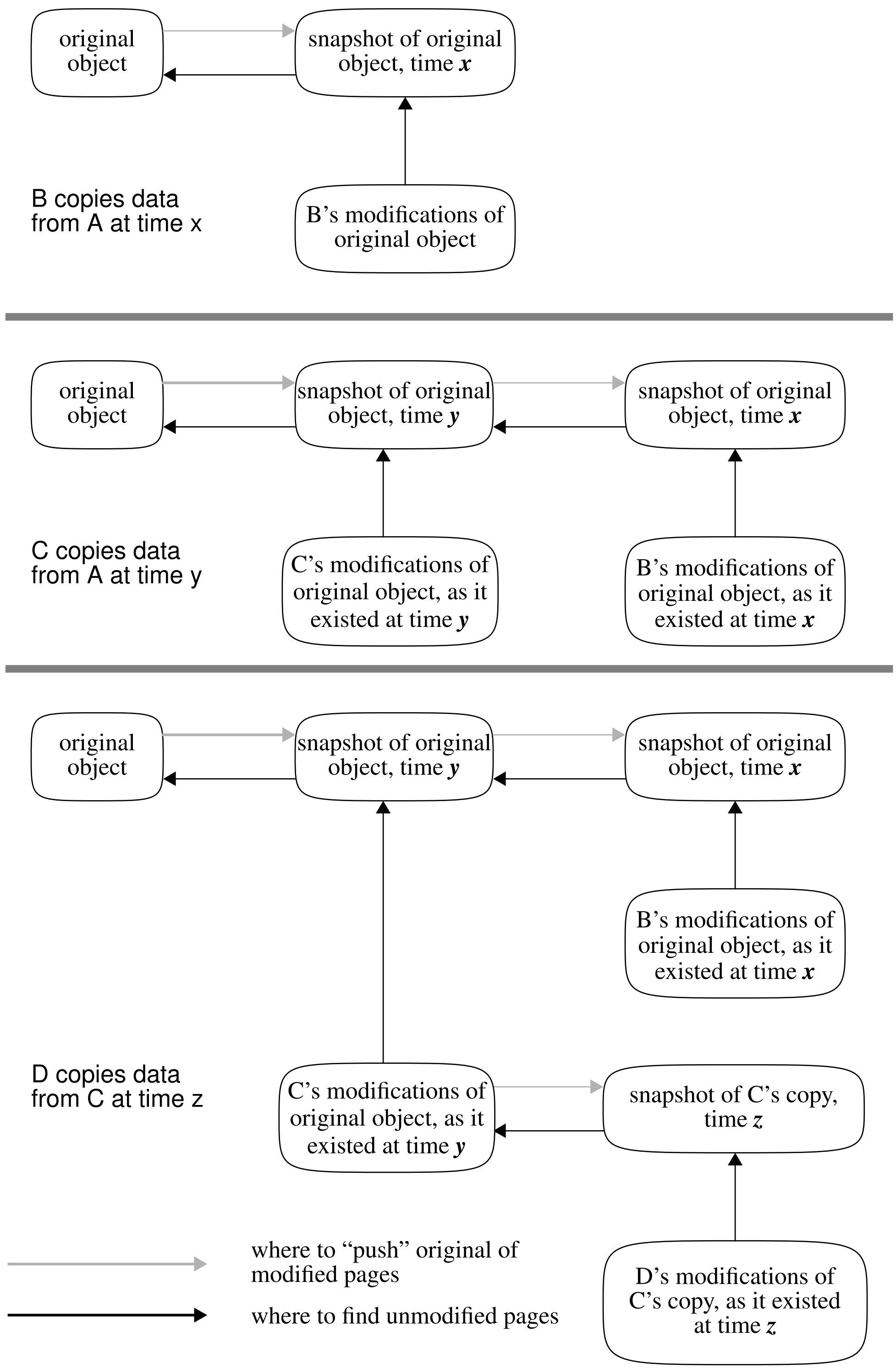


Figure 22 **Virtual Copy Tree Development**

Subsystem Initialization

The initialization of this subsystem consists simply of constructing the virtual_paging object and the two cap_tables employed by the *memory_cache* and *memory_object_rep* classes.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 10

Connection Management Subsystem Design

Overview

A *connection* is an abstract mechanism that represents a communication endpoint. There are two main types of connections: *user ports* and *kernel objects*, and a lessor one, *discard* (which simply discards any message sent to it (by sending to the dead letter office)). The Connection Management subsystem manages the kernel's global *connection name spac*e and implements mechanisms for mapping a capability, via its connection, to the appropriate communication endpoint.

The Connection Management subsystem implements all services associated with the mapping of capabilities to connections including the following:

- For message send operations, the subsystem locates the appropriate connection.
- For message receive operations, the subsystem locates the appropriate transfer queue associated with the *user port* or *port set*.
- The subsystem responds to queries about a given connection's state (whether or not the connection is alive).
- Upon request, the subsystem will send a task a *dead name notification* upon the death of a particular connection.
- Each connection represents some entity that consumes system resources; the subsystem implements and exports *resource accounts* that allow higher level resource management mechanisms to track the resources consumed by related groups of connections.
- The subsystem names the *dead letter office,* the destination of all messages that are sent to dead connections.

Capabilities

The Connection Management subsystem is invoked primarily via capabilities. A capability is a globally unique name that is associated with one and only one connection. A capability retains validity even after the death of its connection. Messages may be sent via the capability after the connection's death without error, but such messages are delivered to the dead letter office and disposed of accordingly.

Kernel Objects

The Connection Management subsystem exports the *kernel_object* class, the base class from which all first-class kernel objects are derived. All first class kernel objects are composite objects; they contain objects that are implemented in layers above the Connection Management subsystem. For example, the top object of a thread is its thread object, which is implemented and managed by the Resource Management Objects subsystem. Similarly, the top object of a clock is a kernel_clock object, which is implemented and managed by the Clock Management subsystem.

Invocation of all first class kernel objects is performed exclusively by the Resource Management Objects subsystem because it is in the highest layer of the kernel's object hierarchy and can invoke all subsystems beneath it. The Connection Management subsystem participates in the invocation of a given kernel object by using the capability to the kernel object to locate and lock the kernel object; this done, the Connection Management subsystem returns a pointer to the kernel object and the Resource Management Objects subsystem uses this pointer to invoke the object.

User Ports

User ports are connections between user tasks. A message sent via a user port is placed on the *transfer queue* associated with the port; it is removed from this queue when the task holding the port's receive right receives the message. (If the user port is destroyed before the message is received, the message is sent to the dead letter office to be discarded.)

Port Sets

The Connection Management subsystem provides support for port sets. A port set is an object that enables a task to receive messages from multiple ports via a single user port name. The only visible property of a port set is the transfer queue associated with it.

The subsystem provides methods for moving individual user ports in and out of port sets. These methods are invoked via the capabilities that represent the user ports.

Connection Agents

When a message is sent via a capability, the underlying connection must be locked in order for the operation to proceed. This is a blocking lock and so cannot be held across kernel invocations or taken at interrupt level.

There are places within the kernel that must be able to send messages reliably without having to acquire a connection's blocking lock. For example, the Clock and Device Services subsystem must be able to send alarm messages and I/O completion messages from interrupt level. Such messages are sent via connection agents.

The Connection Management subsystem includes a method for binding and unbinding connection agents to and from connections. When bound to a connection, a connection agent always delivers messages. If the connection is dead when the message is sent, the connection agent forwards the message to the dead letter office.

Although connection agents are bound and unbound via methods exported by the Connection Management subsystem, connection agents themselves are implemented within the Transfer Management subsystem.

Resource Accounts

Each connection, whether a user port or a kernel object, represents an entity that consumes system resources. The kernel's resource management mechanisms must be able to track the consumption of resources by related objects so that the resources can be reclaimed systematically.

The Connection Management subsystem implements and exports *resource accounts*, each of which tracks the resource consumption of a related group of connections. Since each connection is named by a capability (the receive right capability in the case of a user port, and possibly multiple capabilities in the case of some kernel objects), a resource account names its associated connections via their capabilities.

Dead Letter Office

The dead letter office is the kernel's facility for disposing of messages that are sent to dead connections, and messages that were undelivered when their connections were destroyed. The actual destruction of dead letters is performed by the Resource Management Objects subsystem.

Subsystem Interface

The following *capability* methods can be applied to any type of connection, although not all are meaningful for all types.

- *capability*::**access_name** — Create a capability allowing a particular access mode to the target of this capability.
- *capability*::**add_connection_agent** — Associate the given connection_agent with a transfer_route so as to guarantee message delivery via that agent. If the capability names a live user port, this will associate the agent with that port's transfer_route; otherwise it will associate the agent with the dead letter office.
- *capability*::**create_dname** — Create a unique capability whose effective target is already dead.

- *capability*::**get_access** — Determine the kernel access modes the capability grants to the target connection.
- *capability*::**get_attributes** — Return the attributes of the target user port.
- *capability*::**get_resource_target** — Return a locked pointer to the connection. This operation is similar to **get_send_target** except that it returns failure if the target object does not exist.
- *capability*::**get_send_target** — Return a locked pointer to the connection, as well as the connection's target type, access and transfer limits. If the connection names a kernel object, the caller can specify the required type for the operation to succeed; this ability protects the kernel against possible deadlock. If the capability is dead, the returned connection is the "dead" connection.
- *capability*::**get_type** — Determine the object type of the target (assuming it is not dead).
- *capability*::**hash** — Return a hash value in a given range. This utility function is provided for higher subsystems which use capabilities as indices in hash tables.
- *capability*::**is_dead** — Return true if and only if the capability names a dead object.
- *capability*::**request_dname_notif** — Request a dead name notification for the connection's target, using the given capability as the send-once right for the notification. If the connection's target is already dead, send a dead name notification immediately to the send-once right provided.
- *capability*::**send_port_deleted_notif** — Send a port deleted notification to the capability's target.
- *capability*::**set_attributes** — Set the attributes of the target user port as specified.
- *capability*::**target_lock** — Locate the port_lock for the target connection.

The *capability* class exports the following member functions, which can only be applied to user port targets.

- *capability*::**create_user_port** — Create a user port, making this capability name it.
- *capability*::**destroy_rcv** — Destroy the user port.
- *capability*::**destroy_sonce** — Inform the target connection that a send-once right to it has been destroyed without being used, thereby generating a send-once notification.
- *capability*::**move_member** — Move the user port into the specified port set (removing it from its current port set). If no new port set is specified, the user port will no longer reside in any set.
- *capability*::**move_rcv** — Register the movement of the user port's receive right. This removes the user port from its current port set or interrupts any threads receiving directly from the port.
- *capability*::**receive** — Locate and receive from the transfer_queue associated with a user port connection.
- *capability*::**rename_port** — Assign a new user port name to the user port (called when the receive right is moved into a new task's port name space).

The following methods serve special purposes:

- *capability*::**lock_discard_connection** — Locate and lock a discard connection. This method is used only when the kernel holds a name it knows that names (or at least did name) a discard connection. The method succeeds only if the target is not dead.
- *capability*::**lock_kernel_object** — Locate and lock a kernel object. This method is used when a capability should (according to the user) name a kernel object of a given type, but may not. This method succeeds only if the target is not dead and is of the correct kernel object type.
- *capability*::**lock_kernel_object** — Locate and lock a kernel object. This method is used only when the kernel holds a name it knows that names (or at least did name) a kernel object of the correct type. This method is used when the kernel held a kernel object before, but had to release a lock and later re-find the object, given a connection name. Another flavor of this method locates and locks what is believed to be a kernel object of a given type.

The *message_right* class implements kernel message descriptors used to hold rights in transit; it exports the following member functions:

- *message_right*::**create_dname** — Initialize the descriptor to name a receive right to a unique dead capability.
- *message_right*::**create_rcv** — Initialize the descriptor to name a receive right to a newly created user port.
- *message_right*::**destroy** — Destroy the message right, informing the target connection is appropriate.

Most operations upon connections are performed within Connection Management via a supplied capability. The **get_send_target** and **get_resource_target** functions, though, do return a pointer to the connection for the sake of a limited number of operations that must be invoked from or performed within higher layers. One of these is the virtual **send** method. The processing for *kernel_object*::**send** is performed in a higher layer (since the actual target kernel objects are in higher layers). Likewise, *kernel_object*::**resource_destroy** is a higher layer virtual function. User port processing, though, is within Connection Management. The following *connection* methods are exported.

- *connection*::**control_port** — Return the control access capability for this connection.
- *connection*::**get_account** — Returns the resource account associated with this object.
- *connection*::**get_type** — Return the connection type.
- *connection*::**lock** — Lock this connection.
- *connection*::**unlock** — Unlock this connection.
- *connection*::**within_limits** — Indicate whether transfer limits have been exceeded.

The *connection* class also exports the read-only instance variable *control_name* which is the control-access capability for the connection.

- *user_port*::**direct_send** — Nonvirtual version of send, used when it is determined that a user port is involved.
- *user_port*::**resource_destroy** — Destroy this port resource.
- *user_port*::**send** — Send a message through the underlying transfer_route. The user port must be locked on entry; it is unlocked on exit.

The *kernel_object* class exports the following member functions (explained under concurrency control):

- *kernel_object*::**kernel_object** — Construct a kernel_object, associated with a specified resource_account, assigning to it an appropriate set of capabilities. (Most kernel objects have only a single control capability but some objects, such as a ledger, have multiple capabilities — control, name, etc.) The object (its underlying port lock) is locked on return.
- *kernel_object*::**~kernel_object** — Destroy the object, breaking the connection of all capabilities to it. The object must be locked on entry.

The template class *recycling_msg* is a special class of message that recycles, upon attempted deletion, to its parameter kernel object. For this purpose, *kernel_object* defines a virtual **recycle_message** method. Being a template class, recycling_msg's are not considered to be bound to this layer, but are instead bound to the instantiating layer. In this document they are listed in each affected subsystem. All recycling_msg's have as their allocation record the class *recycle_message_record*.

- *recycle_message_record*::**recycle_message_record** — Constructor, records the target kernel object capability.
- *recycle_message_record*::**~recycle_message_record** — Destructor.
- *recycle_message_record*::**disable** — Sets the target capability so that future destructions will not recycle the message.
- *recycle_message_record*::**get_port** — Returns the target capability.

In general, Connection Management transparently forwards items to the dead letter office. The following items are exported for higher layer use of this office.

- *connection*::**get_discard_capability** — Generate a capability naming the dead (discard) connection.
- *connection*::**get_discard_target** — (static member function) Returns a pointer to a discard connection representing the dead letter office (locked). This is used when a higher layer discovers that it holds a capability which may or may not be dead, but whose right type is wrong, so the liveliness of the capability is irrelevant. Since the right cannot be used for sending a message, a dead destination is needed to forward the message to the dead letter office.

The *discard_connection* class is a special connection class that destroys all messages sent through it. This class exports the virtual **send** and **resource_destroy** methods via its derivation from *connection*.

The *dead_name_notification* class handles the messages and supporting information needed to track a dead name request. It exports only its destructor; disposal of the send-once right implied in the contained capability is the responsibility of the caller.

The *port_set* class exports the following member functions. (Most port set manipulation is done by the contained ports.)

- *port_set*::**port_set** — Construct a port_set, named as specified and billed to a given space_source.
- *port_set*::**~port_set** — De-allocate this port_set, removing all ports from it, interrupting any threads receiving from it.
- *port_set*::**receive** — Locate and receive from the underlying transfer_queue.

The *resource_account* class, which derives from the more primitive *space_account*, keeps track of the list of all billable resources (represented by connections). The principle purpose for this class is to support an iterator over the list of connections.

- *resource_account*::**resource_account** — Construct the account as empty.
- *resource_account*::**~resource_account** — Destruct the account. It must be empty.
- *resource_account*::**iteration_init** and *resource_account*::**iteration_next** — A resource_account maintains (coherently) a single iteration point useful for iterating over the elements of the account. The caller of these iterator functions must hold a higher lock to prevent other threads from simultaneously using these functions. Each call returns a connection billed to the account.
- *resource_account*::**kernel** — Static member function that returns a pointer to the kernel's internal resource account used when creating internal kernel objects.

Finally, the *connection_management* class (with only a constructor for initialization) represents the subsystem as a whole.

Table 12 shows the inheritance relationship for this subsystem's classes.

Table 12 Connection Management Class Inheritance

Class	Inherits from
kernel_object	connection
connection	space_source (Space acct)
dead_name_notification	message (IPC xfer)
discard_connection	connection
recycle_message_record	arbitrary_space_record (Space acct)
recycling_msg	message (IPC xfer)
resource_account	space_account (Space acct)
user_port	connection transfer_route (IPC xfer)

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Resident Memory Management for memory allocation.
- Transfer Management for *transfer_route* and *transfer_queue* objects to which it attaches, and for delivering notifications.
- Space Accounting as the basis for recording accounting information.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (ports).
- Provides system facilities used to represent objects and subjects (ports).
- Provides system resources which are used to implement objects and subjects (ports).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 13 shows the object instance relationships for this subsystem.

Table 13 Connection Management Object Relationships

Object	References
connection	dead_name_notification resource_account mutex (Sched)
dead_name_notification	connection_agent (IPC xfer) space_source (Space acct)
discard_connection	transfer_route (IPC xfer) transfer_queue (IPC xfer)

Table 13 Connection Management Object Relationships

Object	References
port_set	user_port transfer_queue (IPC xfer) mutex (Sched)
recycle_message_record	capability
resource_account	connection
user_port	port_set

The *connection* class serves a variety of purposes.

- It derives from *space_source*, representing a billable resource linked against a resource_account. A resource_account object maintains a list of associated connection objects.
- It provides a place to record notification requests.
- It provides a basis for derivation of *user_port* and *kernel_object*, defining virtual functions to be called both internal to and external from Connection Management.

The *user_port* class derives from *connection* and *transfer_route*.

The *kernel_object* class acts as a base class for any kernel entity to which the user may send a message (e.g. *task*, *thread*, *host*). It simply derives from *connection*.

The *port_set* class manages (links together) multiple user_port's, allowing a user thread to receive from all of them at once (multiplex).

The *dead_name_notification* class, private to this subsystem, is used to remember a user request for a dead name notification. These are linked in a list from the associated connection.

This subsystem maintains a single instance of the *discard_connection* class which represents the dead letter office. It is this object that is returned by **get_dead_target**.

Concurrency Issues

A capability is simply an identifier that indirectly names a port; there are no operations that modify capabilities. Whatever synchronization protects the structure containing a capability protects the capability.

Connection Death

Many of the concurrency issues associated with the Connection Management subsystem involve the life and death of connections. In general, a connection (and the derived user port or kernel object) is protected by a port lock, created when the capability naming the connection was generated, and managed by the capability as it manipulates the connection. This simple management is complicated, though, by dead name notification processing and resource accounting.

The purpose of the resource account is to permit higher layers to locate all of the resources (connections) associated with the account. The account does this by providing, one at a time, the capability for each connection. For resource reclamation, the caller will invoke **get_resource_target** on each capability looking to destroy it (**resource_destroy**). If **get_resource_target** fails, the caller assumes that the target is dead. It is important that **get_resource_target** fail only if the target really is completely gone (all space returned by deleting all associated data structures and the space for the connection itself returned by the connection's destructor) so that the caller knows that there are no dangling references that would prevent destruction of the resource account. So, the connection must be locked during destruction and the capability to connection linkage broken only after the connection has finished destruction.

The linkage of connections against a resource account is protected by a separate (global) lock. This lock must be held not only for connection creation and termination but also for correct internal operation of the iterator functions.

Dead name notification processing presents another problem. The rules for dead name processing presented at the kernel interface require that both the user port name space and the target connection be locked when a dead name request is being added or removed. It is not possible in general, though, to hold both locks — obtaining both would violate the locking hierarchy. (Locking a connection that happens to be a kernel object locks the object, which may be arbitrarily high in the locking hierarchy.) Instead, the dead name list for the target connection must be found and processed under a lower lock, without obtaining the port lock in the process. This lower lock is the same lock used to protect the resource account linkages. That is, the global resource/dead name lock protects all connections. Creating a connection and destroying a connection requires holding both the connection's port lock and the global resource/dead name lock.

Port Set Synchronization

Operations upon port sets come from two sources: invocations from ports inserting themselves or removing themselves from the set, and external operations performed upon the set. A port_set uses a mutex to serialize the insertion and removal of user ports from its internal list. This mutex does not serialize external port set operations (it is obtained and released, of course, if the external operation affects the port set membership). The external caller of a port set must protect against multiple simultaneous calls (which it can since external access to the port set is always through a port name space, the protection for which can protect the port set).

The locking hierarchy order for user port and port set manipulation is *connection* (*port_lock*) followed by *port_set*.

Subsystem Initialization

Initialization of this subsystem consists of constructing the global resource/dead name lock, the static discard connection (the one returned by *connection*::**get_discard_target**) and the kernel's internal resource account.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 11 Clock and Device Services Subsystem Design

Overview

The Clock and Device Services subsystem provides the principal support for kernel clocks and devices. The clock services implement a platform-independent *realtime clock* and additional clocks as may be configured, along with generic sleep/alarm services.

The device services implement a framework to support the device drivers. In addition, an internal interface is provided for use by protocol-based device controllers.

Clock and Input/Output interrupt handling occurs within the Clock and Device Services Subsystem. Interrupt vectors are initialized during system start-up for each configured device and clock. Initial alarm handling and I/O post-processing occur at the respective interrupt levels and may result in the sending of messages and/or thread resumptions to respond to the completed request(s).

I/O devices are modeled hierarchically to correspond with the physical configuration of the system. The various objects deriving from the *hw_device* class each describe their respective part of that hierarchy and are linked accordingly.

Subsystem Interface

Families of interfaces are described in the sections that follow, which correspond to the abstractions provided by the Clock and Device Services subsystem to higher layers. Each family is presented in one of the sections that follow.

Clock Services

The *hw_clock* class provides the current clock attributes and anchors the linked list of *alarm_request* (and derived class) objects that may apply to the clock. Separate instances of the hw_clock object exist for the following clock types: the *realtime clock* and *timeout clock* (required), *battery clock,* and/or *highres clock* (these types are optional). The *timeout clock* implements timeouts, which are handled by a thread rather than at the interrupt level; this clock is available only within the kernel.

- *hw_clock*::**get_alarm_maximum_resolution** — Return the maximum alarm resolution.
- *hw_clock*::**get_alarm_minimum_resolution** — Return the minimum alarm resolution.
- *hw_clock*::**get_get_time_resolution** — Return the current *get_time* resolution.
- *hw_clock*::**get_map_time_resolution** — Return the current mapped time resolution.
- *hw_clock*::**get_resolution** — Get the resolution of the clock.
- *hw_clock*::**get_time** — Get the current clock time.
- *hw_clock*::**mmap** — Return the page list for mapping this clock.
- *hw_clock*::**set_resolution** — Set the resolution of the clock and recompute any pending alarms.
- *hw_clock*::**set_time** — Set the current clock time and synchronize clock "ticks" to within one half the current clock resolution. Any pending alarms will be terminated, with time-stamps reflecting the former clock value.

Being virtual methods, the above methods are also exported for the *bb_clock*, *rt_clock*, *hr_clock* and *to_clock* classes.

The *alarm_request* class represents the interface and implementation common to all forms of alarm request. The specific types of alarms derive from this base class. For brevity, we will often refer to alarm requests as simply "alarms". An alarm can be created with parameters such as its alarm time (the time at which the alarm "goes off," or expires). These parameters can be modified to permit reuse of the data structure for multiple successive alarms.

An alarm may be absolute or relative. If it is absolute, the alarm time represents an absolute time. If relative, it represents a time relative to the time at which the alarm was set (i.e., the initial occurrence of the alarm set now will be at time (now + alarm_time)). When an alarm expires, it is "processed" at clock interrupt level. The processing is specific to the type of alarm, and is defined by the **process** method. The process method is also responsible for recycling the alarm if it is a repeating alarm.

A repeating alarm recurs periodically with a specified interval between occurrences. An alarm can be set, on at most one clock at a time, and can also be canceled if it is currently set but has not yet expired. A cancellation attempt may fail because the alarm has already expired. (i.e., there is a race between cancellation and expiration.)

When a clock has its time set, the alarms currently set on that clock will no longer expire at the time intended by the client. Absolute alarms are treated differently from relative alarms in this case. Absolute alarms are terminated. Alarm termination is considered to be abnormal, so a different method ("terminate") is used to process terminated alarms. On the other hand, relative alarms have their next alarm time recomputed so that they will go off at the same "real" time that they would have had the clock not been reset.

- *alarm_event_request*::**alarm_event_request** — Constructor.
- *alarm_event_request*::~**alarm_event_request** — Destructor.
- *alarm_event_request*::**set** — Perform set alarm processing on an event by waiting on the event.
- *alarm_message_request*::**alarm_message_request** — Constructor.
- *alarm_message_request*::~**alarm_message_request** — Destructor.
- *alarm_message_request*::**recycle_message** — Called by clock management when the alarm message has been sent and is being recycled.
- *alarm_message_request*::**reply_now** — Sends the alarm reply message, if the message is available, and updates the *message_sent* flag.
- *alarm_message_request*::**set** — Perform set_alarm processing on a message request by setting the alarm time and calling *alarm_request*::**set**.
- *alarm_message_request*::**set_alarm_type** — Sets the alarm type associated with this alarm request, which was originally provided by the caller via the **alarm_set** system call. This value is returned to the caller via the alarm notification message.
- *alarm_message_request*::**take_message** — Returns the alarm message, if present, making it ineligible for further use.
- *alarm_poll_request*::**alarm_poll_request** — Constructor.
- *alarm_poll_request*::~**alarm_poll_request** — Destructor.
- *alarm_request*::**cancel** — Attempt to cancel this alarm (if the alarm is not enqueued on the specified clock, the attempt fails.
- *alarm_request*::**get_clock** — Returns the hw_clock to which this request is currently bound.
- *alarm_request*::**set** — Set this alarm on the specified clock.
- *alarm_request*::**set_clock** — Cancel this alarm and set its clock to the specified value.
- *alarm_request*::**set_time** — Set the alarm time to the specified value, indicating whether this alarm is absolute or relative.

To support the MK++ **clock_sleep** trap efficiently, the *thread_alarms* class contains an alarm_event_request object; this request further derives from the *alarm_request* class but is only linked to the realtime *hw_clock* while in use.

- *thread_alarms*::**thread_alarms** — Constructor.
- *thread_alarms*::~**thread_alarms** — Destructor.
- *thread_alarms*::**get_clock_sleep_alarm** — Return this thread's alarm_event_request alarm.

- *thread_alarms*::**get_depress_abort_alarm** — Return this thread's alarm_depress_abort alarm.

The *thread_alarms* class also provides support for program counter (PC) sampling. The *sample_data* class manages the samples, alternating between two recyclable message buffers.

- *thread_alarms*::**get_sample** — returns the status of PC sampling.alarm.
- *thread_alarms*::**set_sample** — Sets PC sampling status.
- *thread_alarms*::**set_sample_data** — Establishes the sample_data to be used for this thread.
- *sample_data*::**sample_data** — Constructor.
- *sample_data*::**~sample_data** — Destructor.
- *sample_data*::**end_sampling** — Indicate that sample collection has ended and that the *sample_data* should flush the buffer. This is a no-op if the current message is not available.
- *sample_data*::**get_sample_conn_agent** — Locates the target connection agent.
- *sample_data*::**recycle_sample** — Called by clock management when a sample message has been sent and is now being recycled.
- *sample_data*::**set_messages** — Called to initially establish the recyclable messages to use.
- *sample_data*::**setup_sampling** — Called to start sampling.
- *sample_data*::**take_message** — Returns the sampling messages, if present, making them ineligible for further use.

The *clock_services* class represents the subsystem state of clock services.

- *clock_services*::**clock_services** — Initialize clock services.
- *clock_services*::**get_clock** — Returns a pointer to a specified hw_clock.
- *clock_services*::**terminate** — Terminate the clock services subsystem.

The *clock_services* class (subsystem) also exports two static instances, *real_time_clock* and *time_out_clock*, which are the real-time and time-out clocks, respectively.

Device Services

The *hw_device* class and its subclasses both describe the physical Input/Output configuration and provide matching class functions to use it. The list is anchored by the single device_services object, via one or more hw_bus and hw_controller objects.

- *hw_device*::**get_mmap_size** — Return the size of the device's physically mapped area.
- *hw_device*::**mappable_ioports** — Returns true if there are mappable hardware IO ports associated with the device.
- *hw_device*::**mmap** — Call the respective service in the device driver to obtain the address assigned to the offset provided.

- *hw_device*::**submit** — Pass an asynchronous open or data transfer request to the respective device driver service.
- *hw_device*::**submit_status** — Pass a get or set status request to the respective device driver service.
- *hw_bus*::**first_device** — Return a pointer to the first present device on the bus.
- *hw_bus*::**next_device** — Return a pointer to the next present device on the bus.
- *hw_controller*::**first_device** — Return a pointer to the first present device on the controller.
- *hw_controller*::**next_device** — Return a pointer to the next present device on the controller.

The *hw_device* class also exports four read-only instance variables: *unit* (the unit number), *device_type* (the base class), *d_subdev* (the device count per unit) and *name* (the device's name).

On some hardware platforms, direct access to hardware I/O ports is provided, and can be safely exported to user tasks. When this is possible, the *ioport_list* class encapsulates the mechanisms necessary to name the ports and reference hardware access maps.

- *ioport_list*::**ioport_list** — Constructor.
- *ioport_list*::~**ioport_list** — Destructor.
- *ioport_list*::**map** — Provide the task access (via its hardware address translation map) to the associated hardware I/O ports.
- *ioport_list*::**unmap** — Revoke the task's access (via its hardware address translation map) to the associated hardware I/O ports.

The *io_request* class is the basis for all I/O requests from higher layers, which are managed in queues anchored by the respective controller or device object. Two subclasses derive from it. The io_event_request contains an event object to support synchronous requests. At completion, the event is posted to allow the requesting thread to resume execution. The *io_message_request* derives from *message* and is formatted such that it can be delivered as the corresponding response message using its embedded connection_agent. *io_data_message_request* is similar, but includes a (small) data buffer for inline transfers.

- *io_request*::**io_request** — Constructor.
- *io_request*::~**io_request** — Destructor.
- *io_request*::**get_error** — Return the current status of the request.
- *io_request*::**io_done** — Mark the request as done, and complete post-processing.
- *io_event_request*::**io_event_request** — Constructor.
- *io_event_request*::~**io_event_request** — Destructor.
- *io_message_request*::**io_message_request** — Constructor.
- *io_message_request*::~**io_message_request** — Destructor.
- *io_data_message_request*::**io_data_message_request** — Constructor.
- *io_data_message_request*::~**io_data_message_request** — Destructor.

The *io_request* class also exports two read-only instance variables, *request_type*, which identifies the nature of the request, and *io_cookie*, a value provided when the request was created, not referenced by device services, but usable by the request's creator so that it can remember the request's association.

From the *io_message_request* or *io_data_message_request* classes derive the following specific message classes:

- *device_open_reply* (constructor and **io_done**; destructor is implemented in Device Management)
- *device_read_reply* (constructor, destructor and **io_done**)
- *device_read_reply_inband* (constructor, destructor and **io_done**)
- *device_write_reply* (constructor, destructor and **io_done**)
- *device_write_reply_inband* (constructor, destructor and **io_done**)

The *device_open_reply* also exports a read-only instance variable, *kdev_ptr*, not referenced by device services, which is used by device management to remember the associated kernel_device object.

The *device_filter* class derives from the *connection_agent* object class and is used to attach input filters to network devices, which are represented by the *net_device* derivative of the *hw_device*. It anchors a list of message objects which are sent to the associated connection_agent.

- *net_device*::**set_filter** — Establish/remove the specified device filter from the queue of active filters.
- *device_filter*::**device_filter** — Constructor.
- *device_filter*::**~device_filter** — Destructor.
- *device_filter*::**parse_filter** — Validate the filter supplied by the caller.
- *device_filter*::**recycle_packet** — Reattach and re-queue a used filter packet.
- *device_filter*::**take_packets** — Returns cached filter packet messages, if present, making them ineligible for further use.

The *device_services* class represents the subsystem as a whole.

- *device_services*::**device_services** — Initialize device services.
- *device_services*::**first_bus** — Return a pointer to the first present bus in the configuration.
- *device_services*::**next_bus** — Return a pointer to the next present bus in the configuration.
- *device_services*::**terminate** — Terminate the device services subsystem.

Clock and Device Class Inheritance

Table 14 shows the inheritance relationship for I/O device object classes.

Table 14 Clock and Device Services Class Inheritance

Class	Inherits from
alarm_depress_abort	alarm_request
alarm_event_request	alarm_request
alarm_message_request	alarm_request
alarm_poll_request	alarm_request
alarm_sched_request	alarm_request
alarm_sentinel	alarm_request
bb_clock	hw_clock
clock_dynamic_space	dynamic_space (Space acct)
device_filter	spin_lock (Sched)
device_open_reply	io_message_request
device_read_reply	io_messge_request
device_read_reply_inband	io_messge_request
device_write_reply	io_message_request
device_write_reply_inband	io_data_message_request
disk_device	hw_device
hr_clock	hw_clock
hw_controller	hw_device
io_data_message_request	io_message_request
io_event_request	io_request
io_message_request	io_request message (IPC xfer)
ioport_list	mutex (Sched)
net_device	hw_device
rt_clock	hw_clock
scsi_controller	hw_controller
scsi_cpu_device	scsi_device
scsi_device	hw_device
scsi_disk_device	scsi_device disk_device

Table 14 Clock and Device Services Class Inheritance

Class	Inherits from
scsi_tape_device	scsi_device tape_device
tape_device	hw_device
thread_alarms	memory_accessor (VM res)
to_clock	hw_clock alarm_poll_hook memory_accessor (VM res)
tty_device	hw_device alarm_request
watchdog	alarm_poll_hook alarm_poll_request

Security Policy

This subsystem is responsible for maintaining the physical integrity of data on the various storage devices.

Privileges and Capabilities

Though the details of each controller/device interface are private to the specific device driver, the Clock and Device Services subsystem has full access to the host hardware (device registers). The abstractions it exports to higher layers represent the respective device classes, but are otherwise device-independent.

This subsystem requires the services of the following subsystems:

- Resident Memory Management for buffer allocation.
- Transfer Management for enqueing and transfer of messages.
- Processor Scheduling for synchronization and interrupt entry control.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Provides system facilities used to represent objects and subjects (devices).
- Provides system resources that are used to implement objects and subjects (devices).
- Uses privileged hardware features (device control registers).
- Maintains separation of disk partitions.

- Is expected by other TCB components to function correctly.

Functional Decomposition

The Clock and Device Services subsystem can be decomposed into two major portions, Clock Services and Device Services. Each hardware clock is modeled by a hw_clock object. Likewise, the input/output hardware configuration is modeled by the *hw_device* class and its *hw_bus* and *hw_controller* subclasses, of which hierarchically-linked objects are created at system initialization.

Clock Support

Clocks and devices are alike in that processing occurs both in the preemptible “thread context” and at “interrupt level”. User task requests are made in thread context, often causing modifications to data and/or queues that are shared with interrupt handlers. Interrupt (and periodic “poll”) handling may be momentarily suspended by higher-priority interrupts, but is not subject to scheduler preemption.

Each hw_clock object provides the basis for the control and use of a physical clock. alarm_request objects may be queued from each hw_clock. Each “clock tick” or hardware timer runout generates a physical interrupt, which is processed within this module. Any queued alarm_requests that have reached maturity or are due within 1/2 of the current clock resolution are processed in ascending time order by executing the respective *alarm_request*::**process** method.

Request Handling

Thread context clock requests name a hw_clock object and/or an alarm_request object. Actual modification of the clock time or resolution and modifications to the queue of alarm_request objects is performed non-preemptively, under the control of a spin lock embedded in the hw_clock object.

Interrupt Handling

At each clock interrupt, *hw_clock*::**interrupt** is entered via the corresponding interrupt vector. The new clock time is placed in the clock’s mapped page when applicable to the clock that interrupted. If the first queued alarm_request has reached maturity (within 1/2 the current resolution), the queue is processed sequentially. For each mature request, one of several actions being taken, according to the alarm_request object’s derived subclass.

- If the object is an alarm_event_request, the event object that is embedded in the object is posted.
- If the object is an alarm_message_request, the associated message is sent to the designated connection_agent.
- If the object is an alarm_poll_request, the designated alarm_poll_hook::**poll** entry-point is called in clock interrupt context. The alarm request time is incremented by the poll interval and the request is requeued. These requests are initialized by the clock and device drivers and are invisible to higher layers.

- If the object is an alarm_sched_request, the execution_point::**quantum_update** function is called to notify that subsystem of the passage of time. The alarm request time is then incremented and the request is requeued. This request is attached to the *realtime* clock at subsystem initialization and is invisible to higher layers.

Device Support

The physical configuration is polled at system initialization. A hw_bus object is created for each input/output bus on the system. Likewise, a hw_controller object is created for each (potential) multiunit controller. The *hw_controller* derives from the *hw_device* class to allow the controller to be treated as a device when necessary. A hw_device object is created for each device seen on a multiunit controller and for each single-unit device or controller. In the case of mass storage media, a disk_device object represents the entire unit. In the same way, the net_device supports a network port, the tty_device supports a communications device while the tape_device represents a tape drive.

Pointers to the *hw_device* class are used exclusively for references to objects of both the base class and its subclasses. The hw_device object contains the queue pointers for io_request objects, which are also modeled as a base class with derived subclasses. It also encapsulates the I/O ports, if present, which in turn locate port_mapping objects which locate the address_translation_map's providing access.

Device State Request Processing

Requests that set or interrogate device state, rather than initiating data transfer, are treated synchronously. If the device cannot complete the action immediately, the request fails.

Input/Output Request Processing

All device I/O is treated asynchronously and controlled by io_request objects that are queued from the respective hw_controller or hw_device object. For multi-device controllers, such as a SCSI disk/tape/CD-ROM controller, pending data transfer requests are queued under the individual unit (disk_device or tape_device) objects while controller status requests are queued under the hw_controller object.

Interrupt Handling

Most device drivers are interrupt-driven. The address of the device's interrupt entrypoint is identified to Processor Scheduling at initialization, along with the interrupt type and its *spl* level. When an interrupt occurs, the scheduler preserves the hardware state prior to the interrupt and transfers control at the proper interrupt priority level to the corresponding interrupt handler. It is then the responsibility of the device driver to process any interrupting events without blocking. When interrupt processing is done, the driver invokes Processor Scheduling to restore the pre-interrupt hardware state.

Normal interrupt processing requires the device driver to determine which outstanding io_requests have completed. As with the alarm_request, multiple subclasses are derived from the *io_request* class to control post-processing.

- If the object is an io_event_request, the specified event is posted. Any waiting thread is responsible for destroying the io_event_request object.

- If the object is an io_message_request, which is also derived from the *message* class, the object's embedded message is completed and queued using the connection_agent to which it contains a reference.

As queued I/O requests complete, the device driver is responsible to select and start one or more waiting requests.

Periodic Polling

As an alternative to accepting per-event interrupts, a device driver may be polled at a regular interval. Polling support depends upon either a reusable alarm_event_request object that wakes a waiting per-device poll thread to call device_driver::**poll** or an alarm_poll_request object that makes the call directly from within the clock interrupt context.

Input Filter Processing

Remote Procedure Call servers and other tasks that process unsolicited network input establish such connections by calling net_device::**set_filter** to create and attach a device_filter to the respective net_device. Each device_filter has an associated queue of messages, into which selected input packets are copied by the device driver's interrupt handler.

The interrupt handler itself, receives network input packets into a dedicated receive buffer. As each packet arrives, the interrupt handler examines the queue of device_filters for matches. For each matching device_filter, the first message is dequeued and the input data copied into it. The device_filter derives from a connection_agent associated with the requestor's *receive port*, to which the message is enqueued. Once the filters have been examined, the network input packet is discarded and a new read made outstanding.

Proper operation of device filters requires higher layers to recycle the filter messages. If this is not done, the initial queue will soon be exhausted and matching input lost. Rather than imposing a change of user interface upon the (potentially many) users of this functionality, device management supplies recycling_msg's that will reconnect the message to its device_filter (via device management mechanisms).

Device Drivers

Essential support is provided to those device drivers that are depended upon by trusted servers. The general model for such drivers follows that for OSF/1™ device drivers, but subject to additional layering and minimality constraints imposed by the trusted kernel. Both character and block device drivers are supported.

Console Port (Character) Driver

An architecture-specific character driver is provided to support the console port (screen, keyboard, and mouse) on the reference architecture. For the reference Intel 386/486 platform, the console port driver supports both character cell and bitmap-based operations. Monochrome, CGA, EGA, and VGA adapters are supported in both character cell and bitmapped modes, as appropriate to the specific adapter.

Floppy Disk (Block) Driver

An architecture-specific block driver is provided to support floppy disk drives. For the reference Intel 386/486 platform, the Intel 82077 Floppy Disk Controller is supported.

Local Area Network Port (Character) Driver

Architecture-specific character drivers are provided to support configured Local Area Network (LAN) adapters. For the reference Intel platform, the 3COM Etherlink 3C501, Olivetti PC586, and Western Digital 8008E EtherNet™ adapter cards are supported.

Parallel Port (Character) Driver

No parallel port driver is provided.

Serial Port (Character) Driver

An architecture-specific character driver is provided for configured serial port(s). For the reference Intel platform, this driver supports ANSI character cell terminals and serial communications streams only.

SCSI-I Device (Block) Driver

The Small Computer System Interface is supported for hard disk, CD-ROM, and backup media (e.g. tape). It retains the internal structure of the OSF/Mach SCSI subsystem, but with fewer supported adapters. For the reference Intel platform, the Adaptec AHA-15xx and AHA-174x adapters are supported.

SCSI-II Device (Block) Driver

No SCSI-II driver is provided, though existing stub routines are retained. A device driver conforming to the SCSI-II standard would share much of the SCSI-I driver's internal structure in addition to supporting SCSI-II performance and functional enhancements.

Device and Clock Object Relationships

Table 15 shows the object instance relationships for I/O device objects.

Table 15 Device and Clock Services Object Relationships

Object	References
alarm_depress_abort	execution_point (Sched)
alarm_event_request	event (Sched)
alarm_message_request	connection_agent (IPC xfer) message (IPC xfer) spin_lock (Sched)
alarm_poll_request	to_clock alarm_poll_hook
alarm_request	hw_clock
bb_clock	spin_lock (Sched)

Table 15 Device and Clock Services Object Relationships

Object	References
clock_services	hr_clock to_clock rt_clock bb_clock hw_clock
device_filter	net_device message (IPC xfer) connection_agent (IPC xfer)
hw_clock	spin_lock (Sched)
hw_bus	hw_device
hw_controller	hw_device
hw_device	hw_controller hw_bus ioport_list spin_lock (Sched)
io_message_request	connection_agent (IPC xfer)
io_event_request	event (Sched)
io_request	hw_device page_list (VM res)
ioport_list	hw_device port_mapping
net_device	device_filter spin_lock (Sched)
port_mapping	ioport_list address_translation_map (VM res)
rt_clock	page_list (VM res) spin_lock (Sched)
scsi_controller	scsi_device
scsi_device	alarm_event_request
thread_alarms	alarm_event_request alarm_depress_abort
to_clock	rt_clock alarm_event_request spin_lock (Sched)
tty_device	io_request alarm_event_request
watchdog	spin_lock (Sched)

Concurrency Issues

Interrupt processing, including clock "tick" handling, is constrained from taking any blocking locks or performing any actions that might have the effect of blocking the interrupt thread. Therefore, concurrency control over data structures that are shared between interrupt level and thread context must be accomplished by use of unidirectional control fields, spin locks, and atomic queue management/bit test functions provided by lower layers.

Thread context processing, wherein both device I/O and clock/alarm requests are handled, is subject to preemption by higher-priority threads at the same interrupt priority level. Therefore, all queue actions must be performed atomically under the appropriate lock. Queue insertions and deletions occur under spin locks in order to allow for concurrent thread context and interrupt-level actions. Unlocked queue traversals must always proceed in a forward direction in order to assure data consistency in the event of preemption; any queue pointer modifications must assure that no unlocked reader ever sees a deleted or incomplete object.

In the case of periodic polling of an in-kernel device driver that does not have a separate thread for the execution of its polls, the clock interrupt calls the driver directly, but at the (higher) *splclock* level to hold off both clock and device interrupts during execution. Note that this can have the effect of "interrupting" the in-kernel device driver's interrupt entrypoint, if also defined, so special care must be taken to prevent interference. The timeout clock has been implemented to make these calls in thread context and should be used wherever possible.

Subsystem Initialization

The major initialization task for this subsystem is determining the physical configuration. Some of this information is provided by tables generated when the kernel was built. During initialization, the actual hardware state is interrogated to determine all present devices and clocks. For each device, controller or bus, an appropriate object is allocated and constructed.

Once the Processor Scheduling subsystem is called to define the relationships between interrupt types and the interrupt priority level at which each will be handled, interrupts can be enabled.

Languages Employed

Except for a few platform-specific optimizations, this subsystem's functionality is implemented in C++.

CHAPTER 12

Resident Memory Management Subsystem Design

Overview

The Resident Memory Management subsystem manages all aspects of and provides all services related to the system's resident memory. The subsystem provides the following mechanisms and services:

- Mechanisms used by the kernel's other subsystems to dynamically allocate memory for kernel data structures.
- Implementation of the kernel's resident page replacement policy and associated paging mechanisms.
- Implementation and management of each task's *address translation map*, the machine-dependent data structure that the hardware's Memory Management Unit (MMU) uses to perform virtual address translation. This management includes maintaining the consistency of a processor's translation look-aside buffer (TLB).
- Ability to monitor the global state of resident memory to prevent unexpected memory shortages and to manage the flow of large amounts of data through the system so as to minimize the possibility of the system running out of memory.

Dynamic Kernel Memory Allocation

The Resident Memory Management subsystem provides the mechanisms and associated interfaces that the rest of the kernel can use to dynamically allocate memory from the kernel's address space. These mechanisms are used to allocate memory for I/O buffers and for kernel data structures. Memory so allocated is nonpageable by virtue of being removed from the general paging pool lists.

Paging Pool Management

A major aspect of the subsystem is its management of the system's paging pool, that portion of resident memory that is available for the caching of data associated with virtual memory objects. It is this function that consumes the bulk of resident memory usage.

This subsystem manages the allocation of the paging pool's pages, and implements the kernel's page replacement policy and pageout mechanisms. Consequently, this subsystem performs all page-related interactions between the kernel and the system's external memory managers and default memory manager, as specified by the external memory management protocols.

The subsystem issues pagein requests in response to faults taken by threads, but determines independently how many pages to request.

Within this subsystem, each virtual memory object that is currently active is associated with a *resident memory object*. A resident memory object represents those pages of the virtual memory object that are currently resident; the resident memory object also provides the means by which the Resident Memory Management subsystem accesses the backing store associated with the virtual memory object.

There are two types of resident memory objects: *named resident memory objects* and *anonymous resident memory objects*. Each named memory object contains a connection agent that is bound to the virtual memory object's abstract memory object port. The subsystem issues paging requests on the memory object by sending messages via the connection agent.

A *named resident memory object* represents a range of pages in a memory object managed by an external memory manager. When the Resident Memory Management subsystem issues a pagein or pageout request, it specifies the page with a <memory_object,offset> pair.

An *anonymous resident memory object* represents a range of anonymous virtual memory and uses the *default pager* as its external memory manager. The *Resident Paging Object* in the Resident Memory Management subsystem handles the allocation and deallocation of space within available backing storage for anonymous pages. The *anonymous resident memory object* records the secondary storage location of any anonymous page which has been swapped out. The default pager only handles IO to secondary storage; it does not track the location of pages.

Other objects supported by Resident Memory Management include the following:

- *resident page objects* to maintain information about individual physical pages and to enqueue them on various lists.
- *inbound page objects* to represent pages which have been requested from external memory managers but are not yet resident.
- *page lists* which enumerate a set of physical pages usually in transit as out-of-line message data.

Resident Memory Management maintains various queues of resident pages.

- The *active page queue* represents those pages currently mapped or available for immediate mapping.
- The *empty page queue* holds pages which have no valid contents and so are available for immediate allocation.
- Each page on the *free page queue* belongs to a resident memory object, AND the external memory manager has a current copy of it. (Thus the page is unmodified in memory.) Such pages are available for immediate reallocation (because the external memory manager has a copy), but they may also be put back into active use (reactivated) if accessed. Pages are put on this queue when selected for eviction if the foregoing conditions apply. Hence, this queue represents a kind of limbo where pages can be saved from complete eviction if they are reaccessed before being evicted.
- The *candidate page queue* holds pages that are candidates for eviction. The pageout thread keeps pages on this list for a while before evicting them, giving them a chance to be used.
- The *zero page queue* holds pages which, like empty pages, are available for immediate allocation but which have already been zero-filled by the (internal) page zeroing thread.

Address Translation Maps

The subsystem implements and manages the machine-dependent aspects of virtual-to-physical address translation. For each task’s address space, there is an associated address translation map object; this object represents the machine-dependent data structures (page tables) that the hardware’s MMU uses to perform translation and related operations.

As noted above, resident memory objects maintain all information about the correspondence of physical and virtual addresses. Hence, a task's address translation map is effectively a cache of those translations. The kernel caches a given translation in the map only when it is required. The kernel can discard map translations at any time because they can always be restored from the machine-independent objects.

Translation Look-Aside Buffer Consistency

A task’s page tables reside in main memory, but address translation would be prohibitively slow if the MMU had to reference main memory for each translation operation. Most hardware architectures optimize these operations by caching translations in the hardware’s *translation look-aside buffer* (TLB). When a task references a virtual page whose page table entry is not cached in the TLB, the MMU copies the entry from main memory to the TLB. As long as the entry remains in the TLB, the MMU can continue to translate references to the virtual page without having to access the page table entry in main memory.

The resident memory subsystem must ensure the consistency of caches which hold virtual-to-physical translations, i.e. the task's address translation map and the hardware's TLB. Generally, entries in these caches must be updated whenever access to a physical page is removed or lessened. Methods are provided to remove all mappings of a given physical page as well as the mapping of a particular virtual page.

Subsystem Interface

The interfaces to this subsystem divide into three broad categories: dynamic kernel memory allocation, paging pool management and address translation map management.

The class *resident_memory_management* represents the subsystem as a whole.

- *resident_memory_management*::**resident_memory_management** — Constructor.
- *resident_memory_management*::**physical_memory_size** — Return the physical memory size.

Also, the following utility routines are exported:

- **do_ranges_intersect** — Determines if (and where) two address ranges intersect.
- **does_range_contain** — Determines if an address range contains another range (or point)
- **is_valid_address** — Determines if an address value is in the range permitted by the hardware.
- **is_valid_offset** — Determines if an offset value is in the range permitted by the hardware.
- **is_valid_size** — Determines if a size value is in the range permitted by the hardware.
- **size_to_page_count** — Translates a range length into the corresponding number of pages.

Dynamic Kernel Memory Allocation

This subsystem exports a variety of data structure memory allocation routines, mostly in the form of **operator new** and **operator delete**. These operators assume that the allocation to be performed is reasonably sized and guaranteed to succeed. Various special forms are defined associated with special classes that optimize memory management for those situations.

- ::**operator new** (space_source) — Global (general purpose) allocator that allocates space for an arbitrarily sized data structure or array (all arrays use this variant according to C++ language rules), billing the space automatically to a given space_source.
- ::**operator delete** — Global memory de-allocator. Automatically credits the freed space to the associated space_source.

The following classes, present in lower layers, have their memory allocators implemented here: *accounted_space*, *dynamic_allocation_record*, *dynamic_attachment*, *dynamic_space*, *message* and *space_source*.

The memory_accessor class provides the ability to access dynamic memory (especially allocation) and to cache parameters needed to enter a mapping associated with a paging request. There is one memory_accessor per thread that is queued on the thread's address_translation_map whenever the thread is waiting for an inbound page. This class exports the following interfaces:

- *memory_accessor*::**memory_accessor** — Constructor.
- *memory_accessor*::~**memory_accessor** — Destructor.
- *memory_accessor*::**memory_low** — Indicate whether memory is low.
- *memory_accessor*::**memory_low_wait** — Wait until memory is low.
- *memory_accessor*::**memory_reserved** — Determine whether this thread holds reserved memory.
- *memory_accessor*::**privilege** — Make accessor memory privileged.
- *memory_accessor*::**reserve** — Ensure that the thread can allocate the specified amount of memory, waiting interruptably if necessary.
- *memory_accessor*::**unreserve** — Give back any reserved memory that is left.
- *memory_accessor*::**usable_memory** — Indicate total amount of usable memory.

The message_memory class implements the kernel message descriptor used to describe out-of-line memory; this class exports the following interfaces from this subsystem:

- *message_memory*::**make** — Initialize the descriptor. There are two "flavors", one to describe a freshly allocated set of non-zeroed pages, and one to transfer an existing page list.
- *message_memory*::**set_size** — Change the actual size of the message data to the size specified.

The kernel_virtual_memory class provides the means to allocate space within kernel dynamic virtual memory. This class exports the following interfaces:

- *kernel_virtual_memory*::**alloc** — Allocate kernel virtual address space of specified size from the system's object. On return, the allocated range is fully populated and backed by real memory that remains resident. If the size corresponds to a page or more, the returned address is guaranteed to be page-aligned. On exit, the global lock is unlocked. There are two flavors of this method: one as described previously, the other additionally specifies where to bill the allocation.
- *kernel_virtual_memory*::**free** — Release kernel virtual memory previously obtained with **alloc**. On exit, the global lock is unlocked.
- *kernel_virtual_memory*::**stack_alloc** — Allocate kernel virtual address space for a kernel stack. The result is page-aligned and possibly guarded by a leading page. The global lock is unlocked upon exit.
- *kernel_virtual_memory*::**stack_free** — Release kernel virtual memory previously obtained with **stack_alloc**. On exit, the global lock is unlocked.
- *kernel_virtual_memory*::**suggest_resize** — Given an array element size and the existing dimension, suggest the next dimension size up to achieve optimal memory usage.

In general, when memory is desired, what is desired is a (virtually contiguous) range of (dynamic) kernel virtual space. For out-of-line message regions, which include I/O buffers, it is more efficient simply to pass around a list of resident pages (a page list).

- *page_list*::**page_list** — Constructor. There are two "flavors" of this method: one allocates the specified number of pages; the other acquires pages from the current address map.
- *page_list*::**~page_list** — Destructor; frees any owned pages.
- *page_list*::**operator** [] — Locate a specified physical page.
- *page_list*::**operator new** — Memory allocator. Allocates space for a page_list that references a given number of pages.
- *page_list*::**operator delete** — Matching memory de-allocator.
- *page_list*::**get_page_count** — Return the number of pages in this page list.
- *page_list*::**get_size** — Return the size of the page list.
- *page_list*::**map_enter** — Forcibly enter the pages in the list as specified into an address translation map (wired).
- *page_list*::**map_remove** — Remove all list pages from an address translation map.
- *page_list*::**split** — Generate two page lists from this list, splitting at the specified page number, and deleting this list in the process.
- *page_list*::**truncate** — Generate a new page list with only the specified number of pages, truncating the rest and deleting this list.
- *page_list*::**zero** — Zero all pages in the list.
- *page_list*::**zero_fragments** — Zero the portions of the first and last pages that are outside the specified range.
- *resident_memory*::**page_alloc** — Allocate a set of resident page frames, returning the physical addresses. This call may wait (uninterruptibly) for memory to become available.
- *resident_page*::**page_alloc** — Allocate a resident page frames, returning the resident page object.

Paging Pool Management

In support of Memory Extent Management, this subsystem exports the following interfaces:

- *named_resident_memory*::**named_resident_memory** (connection agent) — Constructor. Initialize a resident memory object to represent a named memory object with a specified connection_agent to an abstract memory object port and a given memory cache control port name (capability) used in the external memory management requests.
- *named_resident_memory*::**named_resident_memory** (page list) — Constructor. Given a list of physical page addresses, this function links them as wired and resident pages of the target named memory object.
- *named_resident_memory*::**~named_resident_memory** — Destructor.
- *named_resident_memory*::**am_disconnected** — Determine whether this object is still attached to the pager port.

- *named_resident_memory*::**discard_mappings** — Discard all page mappings; called, for example, when an object is revoked.
- *named_resident_memory*::**disconnect** — Called as part of **memory_object_destroy** processing. This external memory management directive does not actually destroy the memory object structures (which are considered to belong to the mapping tasks) but the call does prevent further paging activity. This call disconnects the memory object from the abstract memory object port.
- *named_resident_memory*::**get_cluster_size** — Return the desired paging cluster size.
- *named_resident_memory*::**get_memory_cache_control** — Locate the associated memory cache control port capability.
- *named_resident_memory*::**memory_object_data_error** — Handle a response to a previous page request that indicates that a data recovery error has occurred. Wake waiting threads and propagate the error.
- *named_resident_memory*::**memory_object_data_supply** — Handle the response to a previous page request that supplies the requested data.
- *named_resident_memory*::**memory_object_data_unavailable** — Handle the response to a previous page request that indicates that there is no existing data available for this page. This implies a page with initial contents of 0, which is created and otherwise established as if supplied.
- *named_resident_memory*::**memory_object_lock_request** — Provide combinations of flush, clean, and/or protect actions as necessary to support the **memory_object_lock_request** kernel interface.
- *named_resident_memory*::**set_cluster_size** — Set the desired paging cluster size.
- *named_resident_memory*::**vm_fault** — This method supports several cases, including simple fault (one memory object involved) and push/pull faults (involving copying of pages to *want* and/or *target* objects in support of virtual copy). The source page is in *this* object. If the page is not resident, it is requested. Locks are released and the thread waits for the arrival of the page. If no push or pull is specified, the page is entered into the map when it is supplied; otherwise, nothing further is done and KERN_SUCCESS is returned. Hence, a second fault will occur for push and/or pull cases and will then find that the page is resident. If the page is resident, a copy is made and placed in *want* if *want* is specified. A copy is made and placed in *target* if *target* is specified. If *target* is specified, the page in *target* is placed in the current map; otherwise the page in this is mapped. Depending on paging policy decisions, pages can be evicted at any time unless they are wired. Hence, any affected page may potentially be nonresident on return.
- *anonymous_resident_memory*::**anonymous_resident_memory** — Constructor. Initialize a resident memory object to represent an anonymous memory range.
- *anonymous_resident_memory*::**anonymous_resident_memory** (page list) — Constructor. Given a list of physical page addresses, this function links the pages as resident pages of the target anonymous memory object. An argument optionally allows the pages to be marked as wired.
- *anonymous_resident_memory*::**~anonymous_resident_memory** — Destructor.

- *anonymous_resident_memory*::**discard_page_range** — Discard pages from the object. The contents of the pages are lost and the backing store is deallocated, leaving the memory object sparsely populated or empty.
- *anonymous_resident_memory*::**page_exists** — Returns true if the specified page(s) exist, i.e. have storage allocated and holding data. Note that for named resident memory objects, all pages (up to the size specified to the constructor) are assumed to exist and will be requested from the external memory manager if accessed.
- *anonymous_resident_memory*::**take** — Move the page images from the source object to the corresponding offsets in the target object; this operation does not fetch the pages if they are nonresident; only the page description is transferred.
- *anonymous_resident_memory*::**vm_fault** — This method supports several cases, including simple fault (one memory object involved) and push/pull faults (involving copying of pages to *want* and/or *target* objects in support of virtual copy). The source page is in *this* object. If the page is not resident, it is requested. Locks are released and the thread waits for the arrival of the page. If no push or pull is specified, the page is entered into the map when it is supplied; otherwise, nothing further is done and KERN_SUCCESS is returned. Hence, a second fault will occur for push and/or pull cases and will then find that the page is resident. If the page is resident, a copy is made and placed in *want* if *want* is specified. A copy is made and placed in *target* if *target* is specified. If *new_owner* is supplied, *target* must also be supplied, and the page in this is moved to *new_owner*. If the source page is nonexistent, it is created and zero-filled, and logical copies are propagated as described above. If *target* is specified, the page in *target* is placed in the current map; otherwise the page in this is mapped. Depending on paging policy decisions, pages can be evicted at any time unless they are wired. Hence, any affected page may potentially be nonresident on return.
- *resident_memory*::**copy_protect** — Change the (system-wide) page protections for the resident pages in support of copy-on-write/copy-on-reference processing.
- *resident_memory*::**disable_shadowing** — Disable shadowing (read shadowing) of a given page.
- *resident_memory*::**get_statistics** — Return virtual memory statistics.
- *resident_memory*::**is_shadowing_disabled** — Return true if shadowing has been disabled on a specified page.
- *resident_memory*::**name_virtual_paging** — Called to name (provide a capability for) the virtual paging port to be used in outgoing anonymous messages.
- *resident_memory*::**page_protect** — Cause all mappings to have their page permissions lowered to the maximum of the access parameter; write permission is removed in preparation for copy-on-write; read/write permission is removed in preparation for copy-on-reference.
- *resident_memory*::**physical_memory_size** — Return the size of physical memory.
- *resident_memory*::**set_default_pager** — Called to provide the name and a connection agent into which anonymous paging requests will be sent.
- *resident_memory*::**unmap** — Called when a particular task no longer has access to a range of memory (but when the range still exists). This call removes the range from the task's address translation map, and causes any thread waiting for a page in the

range to give up (although the awakened threads will probably retry their manipulations).

- *resident_memory*::**unwire_range** — Mark resident pages in the range un-wired.
- *resident_memory*::**wire_range** — Mark resident pages in the range wired.
- *resident_paging*::**memory_object_data_error** — Wake any threads waiting for these pages and give them an error.
- *resident_paging*::**memory_object_data_supply** — Provide data as requested in response to a previous data_request.
- *resident_paging*::**set_recycle_limit** — Specify the desired maximum logical address for backing storage for anonymous pages. Note that this is advisory. If all addresses within the specified limit become allocated, the limit will be automatically increased.

Address Translation Map Management

Largely internal to this subsystem is the *address_translation_map* class, used as the basis for hardware virtual address translation, and functioning as a cache of virtual address translations. Although most operations upon the address translation map are internal to Resident Memory Management, there are a few methods exported to support the direct manipulation of the address space.

- *address_translation_map*::**address_translation_map** — Constructor.
- *address_translation_map*::**~address_translation_map** — Empty destructor.
- *address_translation_map*::**get_resident_count** — Return the number of pages in this map that are resident.
- *address_translation_map*::**map_io_address** — Enter a mapping for a hardware I/O port into the hardware port access map.
- *address_translation_map*::**map_return** — Switch the current address translation map to the specified map, loading the appropriate hardware context; and remove as a guest of the current map.
- *address_translation_map*::**protect** — Lower the protection attribute associated with the specified range of virtual addresses as mapped within the specified address_translation_map.
- *address_translation_map*::**unmap_io_address** — Remove the mapping for a hardware I/O port from the hardware port access map.
- *address_translation_map*::**visit_map** — Switch from the current address translation map to the specified map, loading the appropriate hardware context, and be listed as a guest. Being a guest of the map allows (kernel) access to the specified map; being listed as a guest allows destruction of the map to force the guest to give up access to the map.

Table 16 shows the inheritance relationship for this subsystem's objects.

Table 16 Resident Memory Management Class Inheritance

Class	Inherits from
address_translation_map	address_translation_base (Sched)
anonymous_resident_memory	resident_memory
inbound_page	event (Sched)
memory_accessor	transfer_receiver (IPC xfer)
named_resident_memory	resident_memory
page_list	arbitrary_attachment (Space acct)

Security Policy

This subsystem is responsible for ensuring that the hardware translation tables within a particular address translation map have the access permissions correctly set as supplied by the callers.

Privileges and Capabilities

This subsystem has direct access to hardware memory translation tables and registers, which this subsystem encapsulates.

This subsystem requires the services of the following subsystems:

- Transfer Management for enqueing and transfer of messages.
- Space Accounting for accounting for allocated space.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Provides system facilities used to represent objects and subjects (memory).
- Provides system resources which are used to implement objects and subjects (memory).
- Uses privileged hardware features (processor control registers, memory mapping tables).
- Supports architectural requirements for trust (process isolation).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 17 shows the object instance relationships for this subsystem.

Table 17 Resident Memory Management Object Relationships

Object	References
address_translation_map	pv_mapping resident_page memory_accessor space_source (Space acct) event (Sched) mutex (Sched)
inbound_page	resident_memory
kernel_virtual_memory	zone mutex (Sched)
memory_accessor	resident_page
named_resident_memory	connection_agent (IPC xfer)
page_array	page_data
page_data	inbound_page resident_page event (Sched)
page_list	resident_page
pv_mapping	address_translation_map
resident_memory	inbound_page page_array resident_page connection_agent (IPC xfer) mutex (Sched)
resident_memory_management	resident_page
resident_page	memory_accessor resident_memory anonymous_resident_memory mutex (Sched) event (Sched)
zone	zone_page
zone_page	zone

As discussed earlier, the subsystem interface exposes the *resident_memory* class and derived classes *named_resident_memory* and *anonymous_resident_memory*.

Associated with each resident memory object is a list of the pages that have ever been accessed (evicted pages must still be tracked). Each resident page is represented by a resident_page object. The list of all pages is optimized via page_array and page_data objects.

The external memory management protocol does not permit the kernel to request a page twice with no intervening response from the manager. This subsystem tracks all outstanding page-in requests by representing each request via a inbound_page object.

The resident_paging object tracks the usage of (abstract) anonymous backing storage. This object is exposed so that Memory Extent Management may derive a kernel object from it to which the default memory manager sends messages.

Most activities in this subsystem are initiated externally. The exception is page-out processing, which is performed by an internal thread. The subsystem keeps a pool of free pages to satisfy requests for page allocations. When the number of pages in this list drops below a specific threshold, the internal page-out thread runs to evict some pages. In support of eviction decisions, all resident pages are linked into lists by various criteria (such as time last noticed as used).

The sizing of physical memory is performed by hardware bootstrap. The initialization of this subsystem takes all physical memory that is not already committed and forms a free paging pool from it. Memory that is committed includes the kernel image itself, the primitive bootstrap image which was loaded with the kernel and any hardware defined regions.

Each active address space is associated with an instantiation of the *address_translation_map* class. An address space's address_translation_map contains a pointer (or pointers) to the machine-dependent data structures.

In addition to the public *address_translation_map* class, the Resident Memory Management subsystem implements a private class, *pv_mapping*, to manage operations that affect multiple address_translation_map objects. The subsystem maintains a list of physical-to-virtual mappings for each of the system's resident page objects. A given resident page object's list contains zero, one, or more pv_mapping objects; each entry within the list represents a virtual mapping to the resident page object.

Internal to this subsystem are *zone* objects, which permit the efficient allocation of objects packed into memory pages. The *zone* class allocates whole pages to itself which it divides into fixed sized slots. There is a separate *zone* object for each different size slot used. System configuration tools determine the set of slot sizes for which *zone* objects are established.

If an **operator new** allocates an object of sufficiently large size, it is allocated in a set of virtually contiguous system virtual pages; otherwise, it is allocated in a slot in an appropriate zone. Each zone keeps a free list of empty slots for efficient allocation. Each zone_page tracks its own usage, and returns itself to the general paging pool when all of its slots become unallocated.

Concurrency Issues

Internally, Resident Memory Management uses two mutex locks, *rm_lock* and *rp_lock*, to protect resident memory objects and resident page queues respectively. These locks are not exported.

The resident memory objects exposed at interfaces must be protected by a lock the caller holds. Consequently, any call which may need to wait interruptably (e.g. **vm_fault**) is given the caller lock so that it may be unlocked during the wait. This also means that callers must be organized such that the lock can be unlocked by these calls and the state restored after the return.

Attempting to reserve memory may block due to a shortage in resident memory. In this instance, the requesting thread blocks to wait noninterruptibly. The pageout thread wakes the thread when additional memory becomes available. A caller external to this subsystem can hold locks while waiting for memory. Internal to the subsystem, though, the global paging lock must not be held when entering such a wait because the lock will need to be obtained in order to satisfy the event.

The other wait mechanisms concern waiting for a page to be supplied or for its access modes to be changed by an external memory manager. Since this wait is satisfied by an external event, it must be an interruptible wait, and all locks must be released before the requesting thread can wait. Since all locks will be released, the reason why the caller was waiting for the page may no longer be the case when (if) the page is supplied (or access changed), so the caller must be prepared to re-evaluate the conditions.

When an external memory manager supplies a requested page, all threads waiting for that event will be wakened. Because such waits are interruptible, however, some or all of the threads which originally entered a wait for the page may now be gone. Hence, the supplied page is only recorded in its containing resident memory object, and the wakened threads will fault again if they still want the page.

The **vm_fault** call is the only interface that can cause a thread to enter an interruptible page-related wait state. It is also the only interface that can make a page fully accessible by subsequent caller code.

If **vm_fault** finds the target page in the desired state (i.e. resident and with the requested access), it makes the page accessible and returns success. If the page is not in the desired state, **vm_fault** sends a message to the external memory manager to request whatever is needed to correct the problem and puts the thread into an interruptible wait for the event which represents the response to the request.

If the wait is not aborted, the **vm_fault** call will return success when the page is supplied. Because all locks were unlocked before the wait, the caller must re-lock any applicable locks and reevaluate the current state before modifying any system data. Typically, however, the caller simply returns up to the point where the fault was first taken and allows the reference to be attempted again.

In some cases, the reference will fault again and will call **vm_fault** again, but this time, the page should be in the desired state as a result of actions taken the previous time. This second fault always occurs when a page that is the source for a virtual copy is non-resident.

As an optimization, the case of a simple fault (no copying needed) on a non-resident page makes the page accessible by doing a map enter when the page is supplied but before waking up the waiting thread. This avoids the second fault. However, it requires that care be taken to abort the pending map enter if the thread is interrupted from the wait or if any other changes are made to the virtual memory structures that might invalidate the mapping.

Manipulations of an address_translation_map are protected by a global mutex. Actual manipulations of hardware page table entries are performed by Processor Scheduling's address_translation_base, which handles hardware TLB consistency.

An internal call to *address_translation_map*::**enter** can result in a call to allocate memory; any internal *address_translation_map* call could result in memory being freed. Hence, calls to enter (e.g. in the **vm_fault** path) cannot hold the global page lock (*rp_lock*) and the global resident memory lock (*rm_lock*) must be held throughout.

It is conceivable that multiple threads waiting for memory allocations could cause deadlock. Such a situation would occur if both threads were holding onto memory, if neither thread could make progress until it got memory, and if no memory will become available until one of the threads succeeds in getting memory so that it can complete its operation. In general, this is assumed to never happen; most kernel invocations use memory sparingly, and the rate of operations is throttled if necessary when free memory becomes low.

Allocation within the kernel virtual map and within zones is protected by a global mutex.

Subsystem Initialization

Subsystem initialization consists of the construction of the following objects:

- Free page list.
- Least recently used page list.
- Zone tables.
- Kernel address translation map.

The construction of the free page list requires knowing what memory is free, information recorded in memory by the hardware bootstrap mechanisms.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 13

Transfer Management Subsystem Design

Overview

The Transfer Management subsystem provides the mechanisms used to transfer data between active entities. Any entity that receives such data is referred to as a transfer_receiver; typically a transfer_receiver is a user thread that is running in kernel mode to perform a message-receive operation, but the subsystem also supports the transfer of data to internal kernel threads.

The subsystem supports the enqueuing and dequeueing of transfer_items to and from transfer_queues, and can directly hand-off a transfer_item to a transfer_receiver that is waiting to receive.

Transfer Items

A transfer_item is an abstraction that represents data in transit. The most obvious example of a transfer_item is a message, but the abstraction is also used within the kernel as a means of passing data structures to internal kernel threads for processing.

Transfer items are unique as far as kernel data structures because unlike the other structures, transfer_items can be transferred between active entities. A transfer_item is always associated with some space_source, and its space_source binding (and space accounting) changes as it is transferred from one entity to another.

Transfer Queues and Transfer Routes

The principal object exported by this subsystem is the transfer_queue. A transfer_queue is a communication endpoint; any entity that receives transfer_items receives them from a transfer_queue.

Much of the subsystem's functionality, and all of its synchronization, is rooted in the *transfer_queue* abstraction. The subsystem supports the enqueuing of transfer_items from interrupt context and this functionality dictates the synchronization of transfer_queues.

When an entity invokes a send operation to transfer a given transfer_item, the item's destination is a transfer_route. The transfer_route orders incoming transfer_items and places them on the appropriate transfer_queue. When an item is removed from the transfer_queue during a receive operation, the transfer_route stamps the message with a sequence number.

Each transfer_queue is associated with at least one transfer_route. If the transfer_queue is associated with a port set object, it will be associated with multiple transfer_routes, one for each port that is a member of the port set.

For the most part, transfer_queues and transfer_routes function as the underlying mechanisms that the Connection Management subsystem uses to support the user port abstraction, but not exclusively so. Transfer routes and transfer queues are used by the kernel itself to implement the deferred processing of certain kernel data structures. Such a structure is sent to an internal kernel thread via a transfer_route and the thread eventually removes the structure from the associate transfer_queue and processes it.

Connection Agents

When an entity sends a message via the kernel's conventional IPC message-send operation, it must acquire the appropriate port lock before the message can be sent. This is a blocking lock and cannot be acquired by code running at interrupt level.

The Transfer Management subsystem exports the *connection_agent* abstraction which the kernel uses, with one exception, to send messages, including sending messages from interrupt level. (The exception is that the kernel does not employ connection_agents when sending reply messages in response to direct kernel remote procedure calls.)

The *connection_agent* mechanism works as follows: The Connection Management subsystem binds a connection_agent to a connection, at which point, interrupt code may use the connection_agent to send transfer_items to the connection. Once a connection_agent is set up, messages sent via the agent are guaranteed delivery. Even if the target route is destroyed, the connection_agent will be re-routed to a special transfer_route (associated with the dead letter office) to guarantee message delivery.

Subsystem Interface

Aside from the following, the subsystem also includes the *transfer_management* class (with only a constructor for initialization) which represents the subsystem as a whole.

The *transfer_item* Class

This class exports the following member functions:

- *transfer_item*::**transfer_item** — Constructor — initialize the transfer_item, associating it with a specified space_source.
- *transfer_item*::**~transfer_item** — Destructor.

The *message* class derives directly from *transfer_item*, providing the message buffer space.

- *message*::**message** — Constructor.
- *message*::**~message** — Destructor.
- *message*::**get_header** — Return a pointer to the message header in the buffer area.
- *message*::**get_recipient_name** — Once received, returns the port name of the recipient at reception time as recorded for the transfer_route.
- *message*::**get_seqno** — Once received, returns the sequence number assigned to the message.

Note that the *message* class and its descendents use special memory allocators (implemented in Resident Memory or higher) for several reasons.

- Messages are free form structures (they have a random sized message body) so their size is not that of the class that defines them.
- In various environments, the memory used for messages forms a pool in which the storage is recycled.
- Messages may be sufficiently large that special handling is necessary to avoid running out of memory.

The message_right class implements kernel message descriptors that are used to hold a right in transit. The following *message_right* interfaces can be called in any context:

- *message_right*::**operator==** — Determine whether the rights match.
- *message_right*::**get_disposition** — Return the type of right.
- *message_right*::**get_port** — Name the target capability.
- *message_right*::**init_attributes** — Initialize the descriptor to name a particular type of right (to a null capability).
- *message_right*::**is_null** — Determine whether the right is null.
- *message_right*::**is_rcv** — Determine whether the right is a receive right.
- *message_right*::**is_send** — Determine whether the right is a send right.
- *message_right*::**is_sonce** — Determine whether the right is a send-once right.
- *message_right*::**make** — Initialize the descriptor to name a particular type of right to a capability.
- *message_right*::**make_null** — Initialize the descriptor to name no right.
- *message_right*::**make_send** — Initialize the descriptor to name a send right to a capability.
- *message_right*::**make_sonce** — Initialize the descriptor to name a send-once right to a capability.

A message_right object contains a capability, most of whose methods are implemented in Connection Management, which has the following methods callable from any context:

- *capability*::**operator==** — Determine whether the capabilities match.
- *capability*::**operator!=** — Determine whether the capabilities don't match.
- *capability*::**operator<** and *capability*::**operator>** — Provides an ordering of capabilities for when two objects of the same type need to be locked.
- *capability*::**is_null** — Determine whether the capability is null.
- *capability*::**make_null** — Initialize the capability to name no connection.

The message_memory class implements kernel message descriptors that are used to hold an out-of-line region (a page list) in transit. The following *message_memory* interfaces can be called in any context:

- *message_memory*::**get_memory —** Return the associated page list.
- *message_memory*::**get_offset —** Return the starting offset in the first page of the region.
- *message_memory*::**get_size —** Return the length of the region.
- *message_memory*::**make_null** — Initialize the descriptor to name no right.

The *transfer_receiver* Class

The *transfer_receiver* class is part of the internal operation of Transfer Management. It is an exported class, though, because higher layer notions of threads derive from it while it in turn derives from *execution_point*. This class exports the following member functions:

- *transfer_receiver*::**transfer_receiver** — Constructor — create a new transfer_receiver associated with a given parent (typically a task) space_source and initial machine state. The receiver is placed in the LIMBO state.
- *transfer_receiver*::**~transfer_receiver** — Destructor.
- *transfer_receiver*::**current** — Return a pointer to the currently executing receiver.

The *transfer_queue* Class

This class represents transfer_item objects waiting for transfer_receiver objects and transfer_receiver objects waiting for transfer_item objects; it exports the following member functions:

- *transfer_queue*::**transfer_queue** — Initialize an empty *transfer_queue*.
- *transfer_queue*::**~transfer_queue** — Destroy the *transfer_queue*.
- *transfer_queue*::**cancel_wait** — Wake any threads waiting for a transfer_item, giving them the supplied error *status*.
- *transfer_queue*::**quiesce** — Part of route tear-down. Causes all enqueued messages associated with the route, as well as any attached connection_agent's, to be re-directed to the dead letter office.

- *transfer_queue*::**receive** — Receive a transfer_item. The caller supplies a lock (which protects the existence of the transfer_queue) which is unlocked as part of the operation. The sequence number and error status are also returned.

The *transfer_queue* class also exports the static instance *dead_queue*, the target queue for dead messages.

The *transfer_route* Class

This class exports the following member functions:

- *transfer_route*::**transfer_route** — Construct a transfer_route with a given transfer_queue and space_source.
- *transfer_route*::**~transfer_route** — Destructor
- *transfer_route*::**add_connection_agent** — Add a new connection_agent to the transfer_route.
- *transfer_route*::**assign_to_transfer_queue** — Assign the transfer_route to a new transfer_queue. Called to move a user port between port sets.
- *transfer_route*::**get_queue** — Return a pointer to the transfer_route's transfer_queue.
- *transfer_route*::**quiesce** — Disconnect from all sources, transferring any queued transfer_items and attached connection_agents to a specified transfer_route/transfer_queue.
- *transfer_route*::**remove_connection_agent** — This static member function removes the specified connection_agent from its transfer_route. This is a static function since a connection_agent asking for this to be done races with the transfer_route unlinking it, so the connection_agent cannot know which transfer_route to use.
- *transfer_route*::**rename** — Assign a new mach_port_name_t to the transfer_route.
- *transfer_route*::**send** — Enqueue the given transfer_item onto the transfer_route's transfer_queue. The caller supplies a lock (which protects the existence of the transfer_route) which is unlocked as part of the operation.

The *transfer_route* class also exports the static instance *dead_route*, a target for dead messages.

The *connection_agent* Class

This class exports the following member functions:

- *connection_agent*::**connection_agent** — Construct the object. There are two "flavors" of this method; the first constructs the agent object bound to the dead letter office; the second constructs the object by copying an existing connection_agent.
- *connection_agent*::**~connection_agent**— Destroy the object, breaking the binding.
- *connection_agent*::**handoff** — Send a message through the target transfer_route, performing a processor hand-off in the process.

- *connection_agent*::**send** — Send a message through the target transfer_route.
- *connection_agent*::**operator =** — Assignment operator.

Table 18 shows the inheritance relationship for this subsystem's objects.

Table 18 Transfer Management Class Inheritance

Class	Inherits from
message	transfer_item
transfer_item	arbitrary_space (Space acct)
transfer_receiver	execution_point (Sched)

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Space Accounting as part of transferring an item.
- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (ports, messages).
- Provides system facilities used to represent objects and subjects (ports).
- Provides system resources which are used to implement objects and subjects (ports).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 19 shows the object instance relationships for this subsystem.

Table 19 Transfer Management Object Relationships

Object	References
connection_agent	transfer_route
transfer_item	transfer_route
transfer_queue	transfer_item spin_lock (Sched) event (Sched) mutex (Sched)
transfer_receiver	transfer_item space_source (Space acct)
transfer_route	transfer_queue connection_agent space_source (Space acct)

The *transfer_receiver* class derives from the Processor Scheduling subsystem's *execution_point* class; the *transfer_receiver* class adds fields that allow transfer_items to be handed directly to receivers.

The *transfer_item* class is associated with the following functionality: the ability to act as a queue_element in the transfer_queue class, and the ability to record the transfer_item's current space_source.

A transfer_route also records a space_source for billing messages it has enqueued. Its main state is the transfer_queue association and the message sequence number.

The *transfer_queue* and *connection_agent* classes do not derive from any other class.

The only state maintained within a connection_agent is its transfer_route association.

A transfer_queue has two main state items: a queue of transfer_items waiting for transfer_receivers, and an event which is basically a queue of transfer_receivers waiting for transfer_items.

Concurrency Issues

The transfer management subsystem has a single, global, spl high spin lock which protects its data structures. In general, this lock is held across all operations with operations designed to be short.

A few operations, most notably **assign_transfer_route**, are complex and potentially unbounded. These operations are performed a step at a time, releasing and re-acquiring the interrupt lock after each step. The apparent atomicity of this operation is preserved by temporarily moving the affected messages to an internal queue so that their transfer_route can be atomically re-assigned.

A transfer_item object, when not enqueued, is private to the thread holding it; when enqueued, its fields are protected by the interrupt lock.

The fields of the *transfer_receiver* class are private to itself, except when the thread is waiting for a transfer_item, at which time they are protected by the interrupt lock.

Subsystem Initialization

The subsystem initialization consists of the construction of the interrupt lock and the "dead" transfer_route and transfer_queue.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 14 Space Accounting Subsystem Design

Overview

The full management of kernel memory space consumption (as part of general resource management) must be directed from a kernel layer high enough to do something about kernel space consumption (such as destroying entities consuming excessive space). It cannot be a requirement to invoke that high layer whenever space is needed. A low layer subsystem is needed which doesn't provide management per se but provides the low level accounting of space consumption in support of higher layer mechanisms. This low layer subsystem is Space Accounting.

From the point of view of accounting for space, data structures within the kernel fall into three categories:

- A space_source (or derivation thereof) — A structure that represents the space consumed by a kernel entity (and its associated random data structures) whose lifetime is under user control (via higher layer mechanisms). An allocation of a space_source involves the use of a space_account object which will account for the space source as well as its associated random structures. Higher layers associate a space source with an identifiable user controllable resource (user ports and first class kernel objects).
- A space_account (or derivation thereof) — A structure that keeps track of the total consumption of all referencing space_source objects. Higher layers associate a space account with a ledger.
- A random data structure (that is, neither of the other two categories). An allocation of a random data structure involves the use of a space_source object which will account for its space consumption.

The Space Accounting subsystem provides low level space accounting mechanisms — keeping the record of kernel space consumption for space accounts. The management of space consumption within the kernel consists of the intelligent use of the low level mechanisms throughout the kernel to properly track the consumption of space by individual accounts and higher level mechanisms to query the space records and take appropriate action.

Subsystem Interface

This subsystem exports several classes aside from *space_account* and *space_source* described above. A special set of classes (one of which is *space_source*) have the property that, when allocated, an additional "allocation" record is allocated (before the object proper) whose purpose is to record accounting information and provide dynamic allocation properties (such as changing space source binding or handling arbitrary-size data).

The special allocation types are:

- *dynamic_space* — An allocation unit whose billed size can vary and whose *space_source* binding is variable.
- *dynamic_attachment* — An allocation unit attached to a *dynamic_space*, thereby growing the billed size of the dynamic_space.
- *arbitrary_space* — A *dynamic_space* object that contains an arbitrary sized buffer.
- *arbitrary_attachment* — A *dynamic_attachment* object that contains an arbitrary sized buffer.
- *accounted_space* — An allocation unit billed directly against a space_account.

The types of allocation records are the following:

- *source_allocation_record* — for *space_source* and *accounted_space*
- *dynamic_space_record* — for *dynamic_space*
- *dynamic_attachment_record* — for *dynamic_attachment*
- *arbitrary_space_record* — for *arbitrary_space*
- *arbitrary_attachment_record* — for *arbitrary_attachment*
- *random_allocation_record* — for all other allocations

The exported methods for the *space_source* class are the following:

- *space_source*::**space_source** — Constructor; names the space account to be referenced by the new space source.
- *space_source*::**~space_source** — Destructor; the space source must show no outstanding allocation.
- *space_source*::**credit** — Account for space being freed by a data structure being disassociated with the source.
- *space_source*::**credit_remaining_space** — Account for space consumed by this source (which must be credited before the object can be destroyed).

- *space_source*::**credit_virtual_space** — Account for address space being credited to this source.
- *space_source*::**debit** — Account for the space consumed by a data structure being (dynamically) associated with the source.
- *space_source*::**debit_initial_space** — Account for the space consumed by this source itself (as recorded, but not billed, when allocated).
- *space_source*::**debit_virtual_space** — Account for address space being billed against this source.
- *space_source*::**get_space_account** — Locate the associated space account.

The *space_source* class also exports the static instance *kernel_space* which is used for billing certain kernel internal structures.

The allocation record for a space_source is a source_allocation_record.

- *source_allocation_record*::**source_allocation_record** — Constructor.
- *source_allocation_record*::**~source_allocation_record** — Destructor.
- *source_allocation_record*::**credit_remaining** — Account for space consumed by this source (which must be credited before the object can be destroyed).
- *source_allocation_record*::**debit_initial** — Account for the space consumed by this source itself (as recorded, but not billed, when allocated).

An *accounted_space* is a special object for certain internal kernel uses. It is like a *space_source* in that it bills against a *space_account*, but no space is billed against it, nor can its size vary.

- *accounted_space*::**accounted_space** — Constructor; names the space account to be referenced by the new space.
- *accounted_space*::**~accounted_space** — Destructor; the space must show no outstanding allocation.

In general, a space_account object is manipulated only via an associated space_source object. The following exported methods are defined:

- *space_account*::**space_account** — Constructor; established an empty account.
- *space_account*::**~space_account** — Destructor; the account must be empty.
- *space_account*::**get_memory_entitlement** — Return total memory usage allowed
- *space_account*::**get_memory_ratio** — Return ratio of memory usage allowed
- *space_account*::**get_memory_usage** — Return total memory usage of all associated space.

The *space_account* also exports the static instance *kernel_account* which is used for billing certain kernel internal structures.

The class *dynamic_allocation* is the basis for all dynamically allocatable objects: those of variable size, those whose account billing can change, those that are recycled, etc.:

- *dynamic_allocation*::**allocation_record** — Locate the start of the allocation unit from the start of this object.

The associated allocation record, *dynamic_allocation_record*, exports no methods, but does export the read-only instance variable *object*, which points to the associated raw object storage.

The class *dynamic_space* represents allocation units whose billed size and space_source binding is variable:

- *dynamic_space*::**dynamic_space** — Constructor.
- *dynamic_space*::~**dynamic_space** — Destructor.
- *dynamic_space*::**transfer** — Account for space being transferred to some other space source.

The class *dynamic_space_record* handles the data, associated with all dynamic_space objects, that tracks their effective sizes and associated space_source objects:

- *dynamic_space_record*::**dynamic_space_record** — Constructor.
- *dynamic_space_record*::~**dynamic_space_record** — Destructor.
- *dynamic_space_record*::**transfer** — Transfer accounting to a different source.

The class *dynamic_attachment* represents allocation units attached to a dynamic_space object, thereby growing the billed size of the dynamic_space:

- *dynamic_attachment*::**attach** — Charge additional space.
- *dynamic_attachment*::**detach** — Credit additional space.
- *dynamic_attachment*::**transfer** — Account for our being transferred to another dynamic space.

The class *dynamic_attachment_record* handles the data, associated with all dynamic_attachment objects, that tracks their effective sizes and associated dynamic_space objects:

- *dynamic_attachment_record*::**dynamic_attachment_record** — Constructor.
- *dynamic_attachment_record*::~**dynamic_attachment_record** — Destructor.

The class *arbitrary_space* represents dynamic_space objects that contain arbitrary-sized buffers:

- *arbitrary_space*::**arbitrary_space** — Constructor.
- *arbitrary_space*::~**arbitrary_space** — Destructor.

The *arbitrary_space* also exports the read-only instance variable *buffer*, which locates the buffer.

The class *arbitrary_space_record* supports the allocation of arbitrary-sized dynamic_space objects:

- *arbitrary_space_record*::**arbitrary_space_record** — Constructor.
- *arbitrary_space_record*::**~arbitrary_space_record** — Destructor.

The class *arbitrary_attachment* represents dynamic_attachment objects that contain arbitrary-sized buffers.

- *arbitrary_attachment*::**arbitrary_attachment** — Constructor.
- *arbitrary_attachment*::**~arbitrary_attachment** — Destructor.

The *arbitrary_attachment* also exports the read-only instance variable *buffer*, which locates the buffer.

The class *arbitrary_attachment_record* supports the allocation of arbitrary-sized dynamic_attachment objects.

- *arbitrary_attachment_record*::**arbitrary_attachment_record** — Constructor.
- *arbitrary_attachment_record*::**~arbitrary_attachment_record** — Destructor.

The class *random_allocation_record* supports the allocation of random data.

- *random_allocation_record*::**random_allocation_record** — Constructor.
- *random_allocation_record*::**~random_allocation_record** — Destructor.
- *random_allocation_record*::**reconstruct** — Called by the Resident Memory array resize allocator to reconstruct (and adjust) the allocation record following re-allocation.

Table 20 shows the inheritance relationship for this subsystem's objects.

Table 20 Space Accounting Class Inheritance

Class	Inherits from
arbitrary_attachment	dynamic_attachment
arbitrary_attachment_record	dynamic_attachment_record
arbitrary_space	dynamic_space
arbitrary_space_record	dynamic_space_record
dynamic_attachment	dynamic_allocation
dynamic_attachment_record	dynamic_allocation_record
dynamic_space	dynamic_allocation
dynamic_space_record	dynamic_allocation_record

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This subsystem uses no special privileges or capabilities.

This subsystem requires the services of the following subsystems:

- Processor Scheduling for synchronization.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Implements data structures used in mediation (ports, tasks).
- Provides system facilities used to represent objects and subjects (ports, memory, devices, tasks).
- Provides system resources which are used to implement objects and subjects (ports, memory, devices, tasks, threads).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 21 shows the object instance relationships for this subsystem.

Table 21 Space Accounting Object Relationships

Object	References
accounted_space	source_allocation_record
dynamic_allocation	dynamic_allocation_record dynamic_attachment_record
dynamic_attachment_record	dynamic_space
dynamic_space_record	space_source
random_allocation_record	space_source
source_allocation_record	space_account
space_account	source_allocation_record
space_source	source_allocation_record

Concurrency Issues

Space consumption can be affected by interrupt level activities, so space consumption fields in both the account and the space sources are protected by a spin lock.

Subsystem Initialization

The only subsystem initialization is the construction of the lock protecting the space consumption data.

Languages Employed

This subsystem is implemented in C++.

CHAPTER 15 Processor Scheduling Subsystem Design

Overview

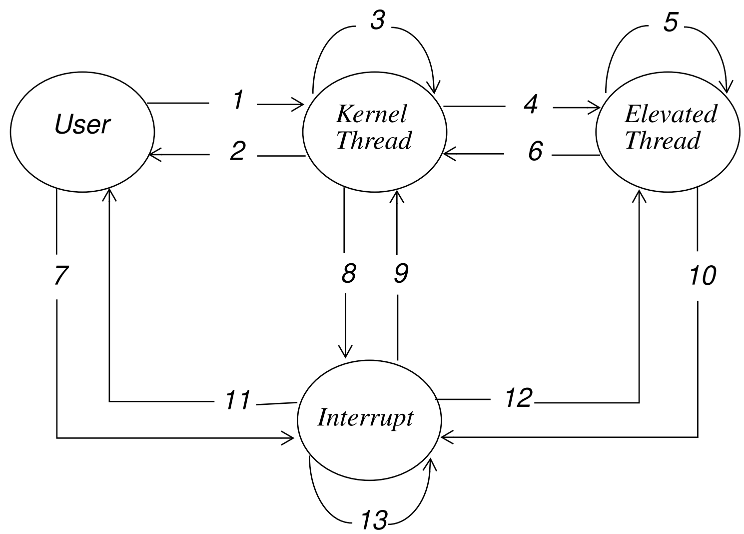
The Processor Scheduling subsystem, the lowest subsystem in the kernel structure, is responsible for controlling in what context each processor is executing at any moment and what activity is being performed on that processor.

This subsystem provides the principal support for managing processing resources by implementing basic *scheduling* and *synchronization* services. The scheduling services multiplex threads of execution, called *execution points*, across available physical processors as well as provide software support for the *interrupt/exception* mechanisms supplied by the hardware. The synchronization services coordinate execution points with internal and external events thus providing some of the basic system concurrency primitives.

Transitions between processor execution context may be initiated by software or hardware. The Processor Scheduling subsystem manages all of the transitions between processor execution context. The machine dependent mechanics involved with these transitions are localized within this subsystem. These transitions are shown in Figure 23.

The Processor Scheduling subsystem itself does not handle interrupts, traps or exceptions, but provides the mechanisms to transfer processor control to higher layer handlers for these events.

The vectoring of interrupts, especially supporting their nesting, involves the hardware notion of the interrupt priority level, which is maintained within the subsystem. Both the interrupt services, and the synchronization services designed to synchronize with interrupt processing manage the interrupt priority level.



1. Trap/Exception

2. *exception_vector*::**return_to_user**

3. Trap/Exception
exception_vector::**return_to_user,** *exception_vector*::**return_to_kernel**

4. *spin_lock*::**lock**

5. *spin_lock*::**lock,** *spin_lock*::**unlock**

6. *spin_lock*::**unlock**

7. Interrupt

8. Interrupt

9. Return from interrupt

10. Interrupt

11. Return from interrupt

12. Return from interrupt

13. Interrupt**,** Return from interrupt
spin_lock::**lock,** *spin_lock*::**unlock**

Figure 23 **Processor Execution Context Transitions**

Kernel Mode Processing and Saving User State

When a user thread requests service of the kernel, the kernel activity is logically considered to be performed by the invoking thread. Occurring on the inside of the kernel protection boundary, this processing is quite different from user processing. The kernel processing has its own stack, its own logical registers, etc. The transition to and from kernel mode involves saving and restoring user state. These operations are implemented within this subsystem.

Processor Set Framework

The scheduling services implement a framework in which physical processors are partitioned into one or more sets of virtual processors. Execution points are assigned to one of the virtual processor sets and may optionally be bound to a specific physical processor within that processor set.

The framework supports the separation of scheduling policy from scheduling mechanism such that different processor sets may employ different algorithms for ordering execution points. The scheduling services transition execution points across scheduling states as shown in Figure 24. The scheduling mechanism supports both *user* and *kernel preemption*, i.e. any transition that makes a more eligible execution point ready to run results in the eviction of some currently active execution point in favor of the more eligible execution point.

Translation Map Management

The preemption of one thread in favor of another involves, among other things, the change in accessible user address space and corresponding change in virtual address translation tables. The tables themselves are maintained by higher layer subsystems. The Processor Scheduling subsystem, however, must be able to name the hardware translation map to use when switching execution point identity. For this purpose, the scheduling subsystem exports the basis for the address translation map, the address translation base.

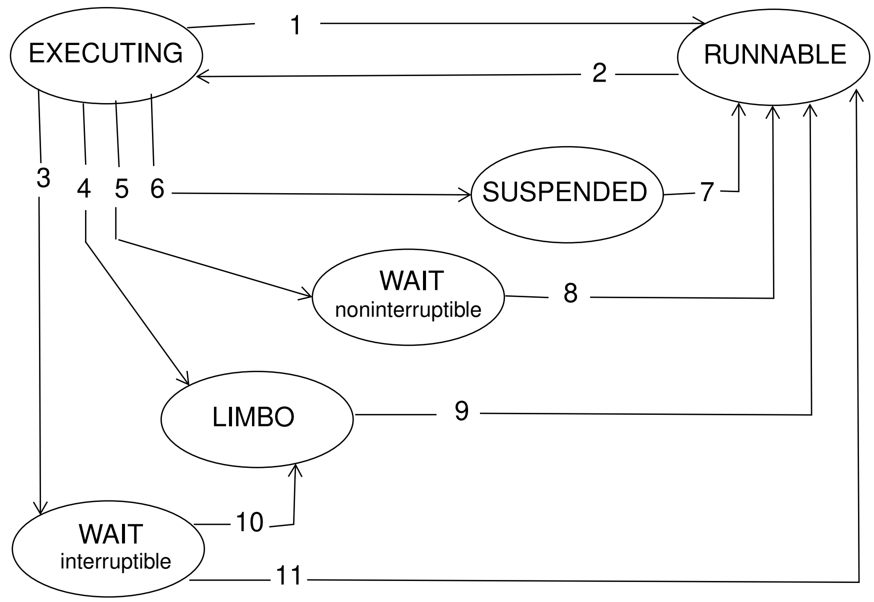
Higher layers maintain the logical relationships between these tables and the Resident Memory structures relating to the pages. The Processor Scheduling's base class, upon direction from Resident Memory, manipulates the actual hardware page tables. This can cause hardware translation caches to become inconsistent, which Processor Scheduling handles by invalidating virtual translation buffers on processors using the particular map.

Synchronization Services

The synchronization services implement various mechanisms to coordinate the execution of threads. These mechanisms include mutual exclusion primitives, called *spin locks, blocking locks,* which arbitrate access to shared system resources; signal/wait primitives, called *events*, which synchronize execution points with internal and external conditions; and connection locking primitives, called *port locks*, which are used solely within the Connection Management subsystem to provide locking and internal location services.

Support for High Level Thread Management

Finally, the Processor Scheduling subsystem provides services in support of higher layer thread management — to support thread suspension, the breaking of interruptible waits, and the synchronization of kernel processing (actually, its completion) with requests to manipulate the thread performing the processing.



1. *execution_point*::**yield,** *execution_point***::depress**
2. Internal scheduler selection routine. This transition is always initiated by another execution point.
3. *event*::**wait** (interruptibly)
4. *execution_point*::**limbo**
5. *event*::**wait** (noninterruptibly)
6. *execution_point*::**suspend**
7. *execution_point*::**setrun**, *execution_point*::**alert**. This transition is always initiated by another execution point.
8. *event*::**signal**. This transition is always initiated by another execution point.
9. *execution_point*::**setrun**. This transition is always initiated by another execution point.
10. *event*::**dq_execution_point**. This transition is always initiated by another execution point.
11. *event*::**signal**, or *execution_point*::**abort**. This transition is always initiated by another execution point.

Figure 24 **Scheduling State Transitions**

Subsystem Interface

The Processor Scheduling subsystem exports the families of interfaces for both the scheduling and synchronization services. Each family will be described in one of the sections that follows.

Scheduling Services

Transitions between processor execution contexts are performed by the Processor Scheduling subsystem (often via hardware mechanisms). In general, these transitions are not atomic — that is, they may be interrupted. As a result, except for the case of interrupting user context, interrupt context cannot meaningfully ask what context it interrupted (it may have interrupted a transition between contexts). Hardware initiated transitions (trap/exception and interrupt) as well as elevating interrupt priority level via *spin_lock*::**lock** can be considered to have been made as soon as the transition is started; the other transitions cannot be considered to have been made until the transition is completed.

Transitions between thread scheduling states, on the other hand, occur completely under the protection of the scheduler's lock and are atomic as seen from outside the scheduler. The transition in Figure 24 from **EXECUTING** to **RUNNABLE** labeled *yield* occurs when an execution point voluntarily gives up the processor (*execution_point*::**yield,** *execution_point*::**depress**) or involuntarily gives up the processor (*execution_point*::**block**) because it was preempted as a side effect of *execution_point*::**quantum_update** or *execution_point*::**setrun**. The transition from **RUNNABLE** to **EXECUTING** labeled *select* occurs when the scheduler selects a thread to run. The transitions to **WAIT** states from **EXECUTING** occur as the result of waiting for an event (possibly a lock); the qualifier in parenthesis indicates whether the wait can or cannot be interrupted (by *execution_point*::**abort**). The transitions from **WAIT** states to **RUNNABLE** labeled *signal* correspond to an event being satisfied (such as *event*::**signal** or *mutex*::**unlock**) or the assertion that the event will not occur (such as destroying an event). The other transitions correspond to particular scheduling interfaces.

The various transitions in Figure 23 and Figure 24 are explained below, in the context of the action or interface call that caused them.

Processor Sets and Execution Points

The *execution_point* class is the abstraction for the basic execution entity in the system, the execution point. It provides both processor and scheduling state for the execution point. There is one execution_point object for each extant execution point. The scheduling state of a thread is explicitly managed by the Processor Scheduling subsystem.

An execution_point has three significant pieces of state from the point of view of users of scheduling:

- The user thread state (as manipulated by the **thread_set_state** and **thread_get_state** kernel calls), meaning the value the processor registers had when the thread last executed in user context and the values they will have the next time the thread executes in user context. The scheduling subsystem guarantees that the user thread state is correct only when one of the following two conditions is true:

- The currently executing thread is manipulating its own state and the processor is in kernel thread context.
- The state to be manipulated is for some other thread and that thread is not in the **EXECUTING** scheduling state.

In all other cases accessing the user thread state may give inaccurate results and changing the user thread state may not have an affect on the target thread.

- A *think* attribute which, when set, forces the thread out of the **SUSPENDED** scheduling state (and prevents it from entering that state), as well as causing any attempt to leave kernel mode to fail. Setting this attribute causes the thread to think about any higher level thread management it should perform upon itself.
- A *give-up* attribute which, when set, forces the thread out of the **WAIT** (interruptible) state (and prevents it from entering that state). If this attribute is set, so will be the *think* attribute.

These two latter *execution_point* attributes support the thread management primitives implemented in higher layers.

- *execution_point*::**execution_point** — Constructor.
- *execution_point*::**~execution_point** — Destructor.
- *execution_point*::**abort** — Set the *give-up* and *think* attributes. When this call returns, it is guaranteed that the target is in kernel mode (with its user thread state appropriately saved), it is not in the **SUSPENDED** or **WAIT** (interruptible) state, and it will not stall indefinitely (enter an interruptible wait) until its *give-up* attribute is reset and it will not exit kernel mode until its *think* attribute is reset.
- *execution_point*::**abort_wait** — Make this execution point runnable if it is in the wait-interruptible state.
- *execution_point*::**alert** — Set the *think* attribute. When this call returns, it is guaranteed that the target is in kernel mode (with its user thread state appropriately saved), it is not in the **SUSPENDED** state, and it will not exit kernel mode until its *think* attribute is reset.
- *execution_point*::**assign** — Assign this execution point to the specified processor set.
- *execution_point*::**current** — This static member function returns the active execution point for the current processor.
- *execution_point*::**current_map** — This static member function returns the active address translation map for the current processor.
- *execution_point*::**depress** — Depress the scheduling priority of this execution point.
- *execution_point*::**depress_abort** — Abort the scheduling depression of this execution point.
- *execution_point*::**get_give_up** — Return the *give-up* attribute for the execution point.
- *execution_point*::**get_sched_state** —Return the scheduling state for this execution point.

- *execution_point*::**handoff** — Transition this execution point to the **RUNNABLE** scheduling state, which may result in the preemption of an active execution point, and make the current execution point wait for the specified event.
- *execution_point*::**has_thread_alarms** —Returns true if this execution point has thread alarm support.
- *execution_point*::**limbo** — This static member function causes the current execution point to transition to the **LIMBO** state.
- *execution_point*::**morph** —Replace the current execution point with this execution point, preserving the kernel stack and address translation map. This is a special purpose method used exclusively during layer initialization at system boot up
- *execution_point*::**quantum_update** — This static member function is used to notify the scheduler that a clock event, typically, but not necessarily, a tick, has occurred. A possible effect of this call is that the currently executing thread will be forcibly preempted when the processor next returns to kernel thread context.
- *execution_point*::**reset_abort** — Reset the *give-up* attribute. The target can once again enter interruptible waits.
- *execution_point*::**reset_alert** — Reset the *think* and *give-up* attributes. The target can once again exit the kernel (and enter interruptible waits).
- *execution_point*::**resume** — Make this execution point runnable if it is currently suspended.
- *execution_point*::**setrun** — Transition the specified execution point to the **RUNNABLE** scheduling state, which may result in the preemption of an active execution point.
- *execution_point*::**suspend** — This static member function causes the current execution point to transition to the **SUSPENDED** state, unless the thread's *think* attribute is set, in which case the execution point remains in the **EXECUTING** state.
- *execution_point*::**update_policy** — Set the scheduling policy for a given execution point to the specified scheduling policy and apply the policy if different than the execution point's current policy. A possible effect of this call is that the currently executing thread will be forcibly preempted.
- *execution_point*::**yield** — This static member function causes the scheduler to re-examine the scheduling eligibility of the current execution point. The current execution point may be preempted because there is a more eligible execution point that is runnable, in which case the current execution point transitions to the **RUNNABLE** state.

The implementation of the following methods is machine-dependent.

- *execution_point*::**get_fpu —** Return the floating point unit information for the execution point.
- *execution_point*::**get_machine_state** — Return the machine user state for this execution point.
- *execution_point*::**get_map** — Return this execution point's address translation map.
- *execution_point*::**get_stack** — Return this execution point's kernel stack.
- *execution_point*::**get_state** — Return the machine kernel state for this execution point.

- *execution_point*::**get_syscall_number** — Extract the system call number from this execution point's user state.
- *execution_point*::**get_user_pc** — Get the program counter in this execution point's user state.
- *execution_point*::**get_user_stack** — Return the user thread's stack pointer (where to copyin system call arguments).
- *execution_point*::**is_null_exception** — Determine whether this execution point's current user state is a null exception. A null exception is a software exception used to effect preemption when the original trap was precipitated by an interrupt.
- *execution_point*::**is_syscall** — Determine whether this execution point's current user state is a system call exception.
- *execution_point*::**set_illegal_instruction** — Change this execution point's current user state to an illegal instruction exception.
- *execution_point*::**set_fpu —** Set the execution point's floating point unit information.
- *execution_point*::**set_pc** — Set the program counter in this execution point's user state to the specified value.
- *execution_point*::**set_sp** — Set the stack pointer in this execution point's user state to the specified value.
- *execution_point*::**set_state** — Sets the machine user state for this execution point.

The *fpu* class maintains the state of the hardware floating point unit (especially in the case of a co-processor) and associated machinery. Aside from entrypoints to deal with machine dependent consistency and fault management, the following general interfaces are exported:

- *fpu*::**fpu** — Constructor.
- *fpu*::**~fpu** — Destructor.
- *fpu*::**get_kind** — Determine the type of floating point unit.

The *sched_proc* class is the abstraction for the basic processing entity on the system. It provides the physical processor attributes and maintains the list of runnable execution points bound to the processor. This class exports the following method:

- *sched_proc*::**current** — Look up the current processor.
- *sched_proc*::**delay** — Busy wait on the current processor for a specified interval.

The *sched_proc_set* class is the virtual processor abstraction. It provides the attributes for a processor set including the scheduling policy and anchors the list of sched_proc objects that belong to the processor set.

- *sched_proc_set*::**get_load_average** — Return the load average for this processor set.
- *sched_proc_set*::**get_load_factor** — Return the load factor for this processor set.
- *sched_proc_set*::**get_nprocs** — Return the number of processors in this processor set.

Address Translation Base

The *address_translation_base* class is the abstraction for the hardware address-translation context of an execution point. The class encapsulates the hardware translation of virtual addresses to physical addresses (page tables, translation look-aside buffers, etc.). The values to be placed into the hardware tables are determined by Resident Memory (class *address_translation_map*); Processor Scheduling performs the actual hardware manipulations. This subsystem is concerned with the primitive state the scheduler must load into the hardware when changing execution points or changing the map base for an execution point and the hardware consistency mechanisms necessary to support Resident Memory's manipulations of the map. The exported interfaces are machine dependent; the following list is representative.

- *address_translation_base*::**address_translation_base** — Constructor.
- *address_translation_base*::**bootstrap** — Initialize kernel map for dynamic memory.
- *address_translation_base*::**copy** — Copy a physical memory page to another
- *address_translation_base*::**current** — Get the active address translation base for this processor.
- *address_translation_base*::**get_kernel_map** — Return a pointer to the kernel's address translation map.
- *address_translation_base*::**get_pte** — Map the given virtual address to its corresponding page table entry in this address translation map. Various interfaces cover other mappings from address to/from translation table locations.
- *address_translation_base*::**kernel_enter** — Map the specified physical address at the specified virtual address in this address translation map with the specified attributes.
- *address_translation_base*::**kernel_remove** — Unmap the specified virtual address range from this address translation map.
- *address_translation_base*::**pte_clear** — Invalidate specified page table entry.
- *address_translation_base*::**pte_clear_reference** — Clear the reference bit from the specified pte.
- *address_translation_base*::**pte_enter** — Enter the specified value in the specified page table entry preserving the specified bits.
- *address_translation_base*::**pte_is_referenced** — Determine whether the reference bit is set for the specified pte.
- *address_translation_base*::**pte_new** — Enter the specified value in the specified page table entry, where it is known not to be in any TLBs.
- *address_translation_base*::**pte_remove_write** — Remove write access from the specified pte.
- *address_translation_base*::**pte_to_pa** — Extract the physical address named by a specified page table entry.
- *address_translation_base*::**switch_map** — Switch from the current address translation map to the specified map loading the appropriate hardware context.
- *address_translation_base*::**to_physical** — Translate a kernel virtual address to a physical address.

- *address_translation_base*::**to_virtual** — Translate a physical address to a kernel virtual address.
- *address_translation_base*::**update_tlbs** — Arrange for the translation lookaside buffers to be flushed on any processor currently using this address translation map.
- *address_translation_base*::**zero** — Zero the physical memory page specified

On machines that support hardware I/O port access, the *address_translation_base* provides access to the underlying hardware access map (*task state segment*).

The map base also exports the following data members:

- *address_translation_base::is_kernel_map* — Read-only member, true if the map is the kernel's map.
- point.
- *address_translation_base::pde_physical* — Read-only member, physical address of hardware the page tables.
- *address_translation_base::pde_virtual* — Read-only member, virtual address of the hardware page tables.

Traps and Exceptions

The Processor Scheduling subsystem provides the mechanisms for transfer of control to the appropriate subsystem in the case of hardware traps and exceptions. A trap or exception is a hardware detected event indicating a failure (not necessarily fatal) in the current instruction stream. The *exception_vector* class is the abstraction used by other subsystems to register and deregister an entry point that is to receive control when a given exception is signalled by the hardware. This class implements the constructor

- *exception_vector*::**exception_vector** — Constructor.

When a trap or exception occurs in user context, hardware causes a transition into kernel mode and the scheduler transitions to kernel thread context, vectoring control to a specified location:

- *exception_vector*::**user_vector** — Supply the location to which control should vector when an exception occurs in user mode. An exception frame, including the exception type and other hardware attributes are pushed onto the kernel stack prior to this transfer.

When the kernel is done handling the exception, the handler calls:

- *exception_vector*::**return_to_user** — Terminate exception processing.

In the typical case, calling **return_to_user** causes the processor to transition from kernel thread context to user context, with the result that the call does not return. However, if the invoking thread's *think* attribute is set (meaning it should not exit kernel mode without examining its thread management state), the **return_to_user** call fails (returns), leaving the processor in kernel thread context. **return_to_user** will continue to fail until the invoking thread resets its *think* attribute.

There is a special *null* exception defined by the Processor Scheduling subsystem, related to interrupt handling. Normally, the occurrence of an interrupt (described in the next section) interrupts the current processor activity with no regard for what that activity is, and the return from that interrupt returns to that previous activity. It is possible, though, for the return from an interrupt to have the side effect of preempting the currently executing thread. To satisfy the requirements for preemption (for the current thread leaving the **EXECUTING** state) discussed above, if the interrupt would have returned to user context, the scheduler must transition the processor to kernel thread context, with the thread in such a state that, when it next runs, it attempts to leave kernel mode and evaluate its thread state. To do this, the scheduler fabricates a special *null* exception, vectoring it as it would any other trap.

When a trap or exception occurs in kernel thread context (traps and exceptions in elevated thread context or interrupt context are fatal), the scheduler vectors control to a specified location:

- *exception_vector*::**kernel_vector** — Supply the location to which control should vector when an exception occurs in kernel mode. An exception frame, including the exception type and other hardware attributes are pushed onto the kernel stack prior to this transfer.

When the kernel is done handling the exception, the handler calls:

- *exception_vector*::**return_to_kernel** — Terminate exception processing.

This call never returns (the processor remains in kernel thread context, returning to the point of the trap or exception).

Interrupts

The Processor Scheduling subsystem provides the mechanisms for transfer of control to the appropriate subsystem in the case of hardware interrupts as well. The *interrupt_vector* object class is the abstraction used by other subsystems to register and deregister types of interrupts and the entry point that is to receive control when interrupts are signalled by the hardware.

When an interrupt occurs, hardware causes a transition into kernel mode (if not already there) and the scheduler transitions to interrupt context vectoring control to a specified location, as set by *interrupt_vector*::**register_interrupt**. An interrupt table entry specifies the interrupt priority level and other machine dependent attributes that apply when an interrupt of the specified type occurs, as well as a location to which control should vector when such an interrupt occurs. An interrupt frame, including the interrupt type, interrupt priority level and other hardware attributes are pushed onto the kernel stack prior to this transfer.

- *interrupt_vector*::**deregister_interrupt** — Remove an entry from the interrupt table for a specified interrupt type.
- *interrupt_vector*::**register_clock_vector** — Set the specified vector to be the real-time clock vector.
- *interrupt_vector*::**register_interrupt** — Add an entry to the interrupt table for a specified interrupt type.

When the kernel is done handling the exception, the handler simply returns.

Figure 23 also shows processor execution context transitions resulting from masking interrupts, that is, going to elevated thread context. These transitions are explained as a consequence of using spin locks, later in this chapter.

Synchronization Services

The abstractions provided by the synchronization services are used by an execution point to coordinate with both external events and other execution points which may be concurrently running on another physical processor. The spin lock and blocking lock abstractions are used to guarantee exclusive access to data structures. The event abstraction is used in conjunction with either a spin lock, blocking lock or both, and an arbitrary data value to implement a multiprocessor safe *sleep/wakeup* protocol. The code fragments that follow illustrate the protocol using a blocking lock:

```
/*
Block for an event
*/
mutex                                   Mutex;
event                                   Event;
Mutex.lock ();
while (!condition) {
/*
        Atomic block and release lock
*/
        Event.wait (Mutex);
/*
        Lock must be reacquired upon wakeup
*/
        Mutex.lock ();
}
Mutex.unlock ();
/*
Signal an event
*/
Mutex.lock ();
condition = TRUE;
Event.signal ();
Mutex.unlock ();
```

Events

The *event* class anchors the list of execution points waiting for a transition on an arbitrary variable, called a *condition*, to be signalled. Each event object specifies the order in which waiters are unblocked when the transition is signalled and whether or not the wait is *interruptible*. There are three flavors of wakeup ordering: EVENT_RR — wake up in scheduling priority order; EVENT_LIFO — wake up the most recent waiter; and EVENT_HERD — wake up all waiters. EVENT_LIFO is intended for cases where it is asserted that all waiters are the same, and it would be okay to starve some of them (those waiting the longest). By choosing the most recent waiter, EVENT_LIFO hopes to select the execution point most likely to have its data still in the processor caches. An execution point can be broken out of an interruptible wait via the **abort** method on the execution point. The event object has no explicit knowledge or control of the condition. The **wait** methods on the event object release the specified lock(s). It is the responsibility of the invoking code to reacquire the lock(s), if necessary, when the wait completes.

- *event*::**event** — Construct an instance of the *event* object class.
- *event*::**~event** —Destroy an event, waking all waiters with an error.
- *event*::**dq_execution_point** — Remove one execution point from the list of waiters. The execution point transitions from **WAITING** to **LIMBO**.
- *event*::**signal** — Signal a condition, unblocking one or all execution points. The execution point(s) transition from **WAITING** to **RUNNABLE**.
- *event*::**wait** — Block the current execution point waiting on a condition protected by lock(s), optionally atomically releasing the lock(s). The execution point transitions from **EXECUTING** to **WAITING**, unlocking the supplied locks.

Spin Locks

The *spin_lock* class is used to arbitrate access to a shared kernel resource. If the desired resource is not currently available, the current execution point busy waits until the resource is available. Each spin_lock object maintains an interrupt priority level attribute. This is a machine dependent value that specifies the interrupt priority level to which the processor is raised while the lock is held.

- *spin_lock*::**spin_lock** — Construct an instance of the *spin_lock* class in the unlocked state. Initialize the interrupt priority level attribute to a specified value.
- *spin_lock*::**~spin_lock** — Destructor.
- *spin_lock*::**lock** — Acquire exclusive access to a resource, busy waiting the current execution point until the resource becomes available.
- *spin_lock*::**lock_interlock** — Acquire the underlying interlock for this lock, busy-waiting until the lock is available.
- *spin_lock*::**unlock** — Release access to a resource, allowing any busy waiting waiters to compete for the lock.
- *spin_lock*::**unlock_interlock** — Release the underlying interlock for this lock.

Transitions to and from elevated context occur as a side effect of using spin locks. If the processor is in interrupt context or elevated context, obtaining (locking) a spin lock and the corresponding release (unlocking) has no effect on the processor execution context. In kernel thread context, however, obtaining a spin lock will transition to elevated thread context and the subsequent release will transition back to kernel thread context. This transition back to kernel thread context may result in the calling thread being preempted; this will just appear to that thread (when next run) as a normal return from the unlock call.

Blocking Locks

The *mutex* class is used to arbitrate access to a shared kernel resource. If the desired resource is not currently available, the current execution point blocks, i.e. gives up the processor and waits until the resource becomes available.

- *mutex*::**mutex** — Constructor — returns with mutex unlocked.
- *mutex*::**~mutex** — Destructor.

- *mutex*::**lock** — Acquire exclusive access to a resource, blocking the current execution point until the resource becomes available.
- *mutex*::**unlock** — Release access to a resource, unblocking one waiting execution point if any are waiting.

Port Locks

The *port_lock* class is used by Connection Management to arbitrate and validate access to connection objects. It is derived from the *mutex* class and additionally provides identity, type and *owner* attributes. Port locks are an abstraction completely hidden behind the *capability* class to map capabilities to connections, locking the connection in the process.

A capability contains a pointer to a port_lock as well as a generation number. A port_lock has the property that once a slot of memory is used for one, that memory will always be a port lock, so the pointer in a capability is always valid and can be followed.

When the connection named by a capability exists, the generation number in the port_lock will match that in the capability; when the connection is destroyed, the generation number is changed.

- *port_lock*::**port_lock** — Constructor called for freshly allocated port locks (when there are no free port locks). There is no destructor since port locks are never deallocated.
- *port_lock*::**change_owner** — Change the target connection for this lock.
- *port_lock*::**find_free** — Called when a new connection is to be created so as to locate a port lock not in use (whose old connection had been previously destroyed) and re-initialize it.
- *port_lock*::**get_access** — Return the kernel capability access to the object on the basis of a supplied capability.
- *port_lock*::**get_type** — Return the kernel object type of the target connection.
- *port_lock*::**identity** — Determine a capability that would name this lock.
- *port_lock*::**lock** — Acquire exclusive access to the associated connection, returning the connection pointer, assuming the supplied generation number matches, blocking the current execution point until the connection becomes available.
- *port_lock*::**lock_typed** —Acquire exclusive access to the associated connection, returning the connection pointer, assuming the supplied generation number and target connection type matches, blocking the current execution point until the connection becomes available.
- *port_lock*::**traverse** — Return a pointer to the associated connection, assuming the supplied generation number matches, without locking the port lock. The caller must hold some higher level lock that protects the port lock to connection binding.
- *port_lock*::**unlock** — Release access to the associated connection.
- *port_lock*::**unlock_disconnect** — Invalidate a port lock, unblocking all waiting execution points with an error.

CPU Usage

The *cpu_usage* class, from which class *execution_point* derives, implements a machine-independent abstraction for the tracking of processor usage by the execution_point. The following flavors of CPU usage are recorded:

- System time — Time spent in kernel mode
- User time — Time spent in user mode
- User idle time — Time spent in user mode at depressed priority

This class exports the following interfaces:

- *cpu_usage*::**cpu_usage** — Constructor.
- *cpu_usage*::**~cpu_usage** — Destructor.
- *cpu_usage*::**add_cpu_time** — Add this object's times data to the times data of the specified target object.
- *cpu_usage*::**get_avenrun** — Return statistics on the average number of runnable execution points.
- *cpu_usage*::**get_cpu_time** — Return processor times for the specified mode; the information is valid for both statistical and accurate timing.
- *cpu_usage*::**get_cpu_usage** — Return scaled processor usage for an execution point.
- *cpu_usage*::**get_load_factor** — Return system-wide statistics on the average load factor.

Scheduling and Scheduling Policies

The class *scheduler* provides the machine-independent abstraction for the scheduling layer. It implements such abstractions as the global scheduler lock and kernel preemption. A single static instance of this class is constructed during layer initialization. This class exports the following interfaces:

- *scheduler*::**avail_cpus** — Return the number of processors available on the host.
- *scheduler*::**check_preemption_on_entry** — Enable kernel preemption for the current processor upon entry into the kernel, possibly causing an immediate kernel preemption.
- *scheduler*::**disable_preemption** — Explicitly disable kernel preemption.
- *scheduler*::**get_default_processor_set** — Return the default processor set.
- *scheduler*::**lock_scheduler** — Acquire the global scheduler lock.
- *scheduler*::**master_subtype** — Return the master processor CPU subtype.
- *scheduler*::**master_type** — Return the master processor CPU type.
- *scheduler*::**max_cpus** — Return the number of processors configured on the host.
- *scheduler*::**restore_preemption** — Enable kernel preemption for the current processor, possibly causing an immediate kernel preemption.
- *scheduler*::**set_clock_resolution** — Set the clock resolution for various classes in the scheduling layer.
- *scheduler*::**unlock_scheduler** — Release the global scheduler lock; preempt if appropriate.

The class *sched_data* abstracts the set of run-time scheduling data per execution point. Class *execution_point* derives from this class which exports the following interfaces:

- *sched_data*::**get_sleep_time** — Return the length of time for which the execution point has been blocked.
- *sched_data*::**get_user_policy** — Return the scheduling policy for this object (in the thread kernel interface canonical form).
- *sched_data*::**total_usage** — Total the processor and scheduling usage for this execution point.
- *sched_data*::**usage_update** — Update the processor and scheduling usage for this execution point if more than a second has elapsed since last updated and it's policy is POLICY_TIMESHARE.

The class *sched_policy* abstracts a specific scheduling policy. Class *execution_point* derives from this class which exports the following interfaces:

- *sched_policy*::**sched_policy** — Constructor.
- *sched_policy*::**~sched_policy** — Destructor.
- *sched_policy*::**operator =** — Assignment (copy).
- *sched_policy*::**get_policy** — Return the scheduling policy type.
- *sched_policy*::**get_user_policy** — Return the scheduling policy for this object (in kernel interface canonical form).
- *sched_policy*::**my_policy** — Return the scheduling policy for this execution point.
- *sched_policy*::**set_policy** — Set the scheduling sub-policy and attributes for a given execution point. Execution points normally inherit their scheduling sub-policy and attributes. The exception is the set of execution points that exist only in the kernel. This method is used to set the policy of those execution points before they are first run.
- *sched_policy*::**set_user_policy** — Set the scheduling policy for this object to the policy specified in the kernel interface canonical form.
- *sched_policy*::**update_policy_limits** — Update this policy's base and limit.

The class *sched_policies* abstracts the set of scheduling policies supported by a processor set; it exports the following interfaces:

- *sched_policies*::**get_base_priority** — Return the base priority for the specified policy.
- *sched_policies*::**get_default** — Return the default policy for this processor set.
- *sched_policies*::**get_max_priority** — Return the maximum priority for the specified policy.
- *sched_policies*::**get_min_quantum** — Return the minimum scheduling quantum for this policy set.
- *sched_policies*::**get_min_timeout** — Return the minimum scheduling timeout for this policy set.
- *sched_policies*::**get_policy_info** — Return processor set scheduling policy information.

- *sched_policies*::**is_member** — Determine whether the specified policy is a member of the policy vector.
- *sched_policies*::**set_default** — Set the default scheduling policy to the specified policy and the base priority to the specified value.
- *sched_policies*::**set_enabled_policies** — Reset the set of enabled scheduling policies based on the specified vector of policies.
- *sched_policies*::**set_limit** — Set the maximum priority for the specified policy to the specified value.

Miscellaneous Services

The subsystem provides a *counter* object which provides a hardware atomic counter.

- *counter*::**counter** — Constructor.
- *counter*::**~counter** — Destructor.
- *counter*::**operator +** — Atomic increment.
- *counter*::**operator –** — Atomic decrement.

The following are non-class items.

- *boothowto* — Read-only, set to indicate boot-time flags.
- *boot_device* — Read-only, set to the name of the boot device.
- *boot_string* — Read-only, set to the operator supplied boot string.
- **halt_all_cpus** — halts the system, forcing a reboot.
- **machine_memory_init** — Machine dependent physical memory initialization.
- **panic** — displays a message on the kernel and then halts the system.
- **printf** — displays a message on the system console using special polled I/O operations, bypassing the normal driver.
- *rebootflag* — Set by **panic** or **halt_all_cpus** to indicate to any interrupt handlers that they cease and desist.

In support of writing to the console (by the primitive **panic** and **printf**), Processor Scheduling contains a limited, polled I/O console driver. That driver exports machine dependent interfaces to Device Services to control interference from the interrupt-based driver.

Finally, Processor Scheduling and the hardware export I/O instructions, interrupt masking instructions and bit manipulation instructions.

Table 22 shows the inheritance relationship for this subsystem's object.

Table 22 Processor Scheduling Class Inheritance

Class	Inherits from
event	wait_queue
exception_vector	spl

Table 22 Processor Scheduling Class Inheritance

Class	Inherits from
execution_point	sched_data spl
mutex	wait_queue spl
port_lock	mutex
scheduler	spl
sched_data	sched_policy cpu_usage
sched_proc	interrupt_vector
sched_proc_set	run_queue sched_policies
spin_lock	spl

Security Policy

This subsystem has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This subsystem uses privileged hardware instructions to switch thread contexts.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This subsystem is protection critical because it:

- Provides system resources which are used to implement objects and subjects (threads).
- Uses privileged hardware features (processor control registers).
- Supports architectural requirements for trust (process isolation).
- Is expected by other TCB components to function correctly.

Functional Decomposition

Table 23 shows the object instance relationships for this subsystem.

Table 23 Processor Scheduling Object Relationships

Object	References
cpu_usage	sched_proc_set
execution_point	event sched_proc address_translation_base
interrupt_vector	spin_lock
mutex	execution_point
run_queue	execution_point
scheduler	sched_proc sched_proc_set execution_point
sched_policies	sched_data sched_policy
sched_proc	execution_point spin_lock
sched_proc_set	sched_proc execution_point spin_lock
spin_lock	execution_point
wait_queue	execution_point

The Scheduling subsystem can be decomposed into two major components, *synchronization* services and *scheduling* services. The operation of port locks was explained when explaining their interface and relationship to capabilities.

Synchronization Services

Wait Queues

The mutex, event and port_lock objects contain a wait_queue object to manage the list of waiting execution points. The *wait_queue* class provides for three optional orderings of execution points. The first option is by scheduling priority. This ordering is global, reflecting both the relative importance of the execution point with respect to other execution points within its scheduling policy as well as the relative importance of the policy with respect to other scheduling policies currently active in the system. This ordering permits the selection of the most eligible execution point when the event signalled satisfies only one waiter, e.g. the release of a shared resource. The second ordering is LIFO. The most recent waiter is selected. This ordering assumes all waiters are of equal importance and therefore takes advantage of hardware side effects such as cache warming. The last ordering is also LIFO, but in this case all waiters are awakened. Waking up all waiters, sometimes called a *thundering herd* wakeup, is done only in the case when the event signalled satisfies all waiters, e.g. a pagein request completion.

Spin Locks

The *spin_lock* class is used to access shared kernel resources. Each spin_lock object implements a busy wait protocol and has an associated machine dependent interrupt priority level. The code paths bracketed by a spin lock are protected against concurrent access by other processors as well as interrupt level code on the processor holding the spin lock. An execution point holding a spin lock may not be preempted and may not block. Code paths protected by spin locks represent non-preemptible *critical regions*.

Spin locks are the only exported mechanism that manipulate processor interrupt priority levels. The association of a processor interrupt priority level with each lock forces a strict locking hierarchy on the abstraction, i.e. nested locks must be acquired in order of increasing processor interrupt priority and must be released in the reverse order. In the case of a uniprocessor implementation, the lock busy wait protocol becomes a null operation. The actual hardware mechanics involved with priority levels is encapsulated in the class *spl*.

Blocking Locks

The *mutex* class is also used to access shared kernel resources. Each mutex implements a blocking protocol in which the current execution point is blocked if the resource is not available. The code paths bracketed by a blocking lock are only protected from concurrent access by another processor. They are not protected from interrupt level code. An execution point may be preempted while holding a blocking lock but should not block indefinitely while holding one.

Scheduling Services

Run Queues

Like the *wait_queue* class the *run_queue* class provides an ordering for execution points. The ordering is local to the scheduling policy reflecting the relative importance of the execution point with respect to other execution points scheduled by the same policy.

Processors and Processor Sets

The *sched_proc* class is the abstraction for the basic processing entity in the system. It provides the physical processor attributes and anchors the linked list of execution_points that are bound to the physical processor. One sched_proc object is statically configured per physical processor at system boot time to provide a basis for the control of the physical processor. One or more sched_procs may be grouped to form a sched_proc_set object. The sched_proc_set enforces a uniform scheduling policy across all processors in the set. By default a single instance of the *sched_proc_set* class is created at boot time with all available sched_procs statically assigned to it. The system may be optionally configured to create multiple sched_proc_sets at boot time and statically partition available sched_procs across the sched_proc_sets. In the presence of a single processor set all execution points are by default assigned to it. The sched_proc_set and sched_proc objects contain a run_queue object which is used to manage its list of runnable execution points. The sched_proc_set object contains a run_queue object which is used to manage its list of runnable execution points. In a multiprocessor configuration, each sched_proc object also has a run_queue.

Scheduling Policy

Each sched_proc_set object observes the scheduling policy implemented by the run_queue object from which it derives. The methods supplied by this class select the most eligible execution point from a list of runnable execution points. The scheduling policy for the default sched_proc_set is a priority based algorithm supporting both timesharing and preemptive real-time sub-policies. This is the only policy provided.

The basic scheduling entity in the system is the execution_point object. Higher level thread objects are derived from this object. There is one instance of the *execution_point* class per extant thread of execution in the system. The execution_point object provides the basis for control of a thread of execution. Physical processor state is maintained by the execution_point object and these objects are scheduled onto sched_procs according to the scheduling policy of the associated sched_proc_set. The only exception to this rule is for execution_point objects whose scheduling priority is depressed. Depressed execution points only run when nothing else is runnable and are not managed by the particular scheduling policy for the processor set.

Preemption and Alert

The Processor Scheduling subsystem implements a *fully preemptive* mechanism for real-time scheduling policies. Execution points are always evicted in favor of a more eligible execution point that has been made runnable, regardless of the scheduling policy assigned to the superior execution point. Furthermore *user* space preemption can occur on any instruction boundary. The mechanics of this involve forcing the processor to interrupt context via some interrupt (possibly the clock), deciding that the interrupted thread should be preempted, arranging thereby, described below, that when the interrupt service routine returns, the scheduler transitions the processor to kernel thread context, changing the running thread in the process.

Preemption is always a transition from one thread executing on the processor with the processor in kernel thread context changing to a different thread executing on the process with the processor still in kernel thread context. Kernel preemption can only happen when the processor is in kernel thread context.

When the current processor should consider preempting the currently executing thread, or another processor should consider preempting its thread or forcing its thread state (satisfy the guarantees of the **alert** function), a *preempt* flag is set for that processor. This flag, which is set and cleared under protection of the global scheduling lock, is checked whenever the scheduler is about to lower interrupt priority level to zero.

Interrupts and Exceptions

A single instance of the *interrupt_vector* class and the *exception_vector* class is statically configured at system boot. The appropriate subsystems may then register entry points to handle each interrupt and exception that may be generated by the hardware.

Concurrency Issues

Hardware Support

For multiprocessor implementations the Processor Scheduling subsystem requires that the following mechanisms be supported by the hardware platform

- Each processor must be able to uniquely identify itself such that the identifier can be mapped onto a range of integers. The integer id is used as an index to lookup such per processor items as the current execution point, i.e. implement the static member function *execution_point*::**current.**
- Each processor must be able to direct inter-processor interrupts to each of the other processors to signal scheduling events.
- The hardware memory subsystem must support some form of atomic memory access on which to build the basic concurrency primitives, e.g. locked bus cycle, read-modify-write, etc.

Locking

The basic locking primitives exported by the Processor Scheduling subsystem, spin locks, blocking locks, and port locks, are built on the atomic memory access support implemented in the hardware. In addition, the subsystem uses these basic primitives internally to ensure concurrency. The *scheduler* class contains a spin lock to protect scheduling state transitions on an execution point.

Preemption and Alert

An execution point is preempted only when the processor is in kernel thread context. For this decision to have been made, it follows that the processor is in kernel mode (possibly processing an interrupt that interrupted user context execution) and that interrupt priority level is greater than zero (because the scheduler internally locks a spin lock when making such decisions). It is when the processor's interrupt priority level returns to zero (the interrupt handler tries to return or the scheduler call involved wants to return) that the scheduler will force the processor to kernel thread context and possibly change thread identities, if the processor's preempt flag is set. This transition is a simple enough matter, if it were not for the guarantees required of having saved thread state for a non-running thread. (Even if a decision is ultimately made not to preempt the thread, these guarantees are satisfied whenever the processor's *preempt* flag is set, to satisfy the guarantees of the **alert** function.)

Satisfying these guarantees is no problem if the processor is leaving elevated context (since the entry to elevated context came from kernel thread context, which satisfied these guarantees when kernel thread context was entered). There is only a problem if the processor is in interrupt context.

If the interrupt interrupted user context, then the entry to kernel mode saved all of the user thread state in its area. Preemption is a simple matter of making it look as if the user thread took a *null* exception (which involves modifying the interrupt frame on the stack to be an exception frame) before switching threads.

The other problem area is if the interrupt interrupted the user-to-kernel-thread or kernel-thread-to-user context transition logic in the scheduler that saves or restores user thread state. The process of returning from an interrupt to interrupt priority level zero checks to see if the activity interrupted was one of these. If the interrupted activity was saving user thread state, the *preempt* flag is left on and the activity resumed; the scheduler's thread saving logic checks the *preempt* flag after saving user state and preempts then. If the interrupted activity was exiting the kernel (which consists of checking the *think* attribute and then restoring the user state and leaving the kernel), the thread is preempted at this time, but the machine state to be restored when the thread next runs is altered so as to perform the complete kernel-thread-to-user transition, including the checking of the *think* attribute.

Note that setting the *think* flag (the **alert** or **abort** functions) is never called from an interrupt service routine. Therefore, the checking of this flag does not race with interrupt servicing on the current processor. It can, however, race with the setting of the flag by thread management primitives invoked on other processors.

To force an **alert** on a thread executing on another processor, that processor's *preempt* flag is set and an interprocessor interrupt sent (under protection of the global scheduling lock). **alert** then loops (not holding the global scheduling lock) until the target processor clears its flag (under protection of the scheduling lock). To avoid deadlock, the processor must check its own *preempt* flag within this loop.

When a processor checks its *preempt* flag it is not under protection of the scheduler lock. This is okay; if the *preempt* flag is not set when the processor checks it, since it is set and then an interrupt sent, it is guaranteed that the target processor will eventually handle an interrupt whose return will notice the flag.

TLB Consistency

Manipulations of an address_translation_base are protected by a spin lock. If the manipulation will cause the map to be changed in such a way as to affect hardware TLB consistency, the TLB must be invalidated.

Because the map being manipulated is often the map in use by the current thread, the thread changes its active map to null. This is interpreted as representing the kernel map but in a special way that won't stall the thread even if the kernel map itself is being changed.

Subsystem Initialization

The process of building the kernel statically configures a number of sched_proc and sched_proc_set objects for the maximum supported configuration for the build. At subsystem initialization, the actual hardware configuration is examined to determine which objects will be used. The constructors for these objects are executed.

The single interrupt_vector and exception_vector are constructed as well. Interrupts are still disabled on the processors at this time.

An empty free list of port locks is constructed.

Languages Employed

The state transition logic in this subsystem is coded in assembly language. General scheduling logic is implemented in C++.

CHAPTER 16

Language Support Library Design

Overview

Since the C++ language inherently defines only very basic data types, C++ programs typically utilize library routines to manage character strings, simple queues, etc. A restricted subset of the ANSI C (C++) library routines are provided for kernel use, as well as some simple data structure classes.

Library Interface

The following ANSI C functions are provided.

- **max**
- **min**
- **memcpy** (and **memcpy16**)
- **memset**
- **strcat**
- **strchr**
- **strcmp**
- **strcpy**
- **strlen**
- **strncmp**
- **strncpy**

The parameterized class *queue*<*base*> defines a doubly linked list of elements; the corresponding parameterized class *queue_element*<*base*> defines an element in such a list. These classes provide the following methods:

- *queue_element*<*base*>::**queue_element** — Constructor
- *queue_element*<*base*>::**~queue_element** — Destructor
- *queue_element*<*base*>::**queue_next** — Returns a pointer to the next element in the queue.
- *queue_element*<*base*>::**queue_prev** — Returns a pointer to the previous element in the queue.
- *queue*<*base*>::**queue** — Constructs an empty queue.
- *queue*<*base*>::**~queue** — Destructor.
- *queue*<*base*>::**append** — Appends an element after a specified one.
- *queue*<*base*>::**dequeue_head** — Removes and returns the first element of the queue.
- *queue*<*base*>::**dequeue_head_known** — Optimized version of **dequeue_head** when it is known that an element exists.
- *queue*<*base*>::**dequeue_tail** — Removes and returns the last element of the queue.
- *queue*<*base*>::**dequeue_tail_known** — Optimized version of **dequeue_tail** when it is known that an element exists.
- *queue*<*base*>::**enqueue_head** — Adds a new element to the front of the queue.
- *queue*<*base*>::**enqueue_tail** — Adds a new element to the end of the queue.
- *queue*<*base*>::**insert** — Inserts an element before a specified one.
- *queue*<*base*>::**queue_empty** — Returns true if the queue is empty.
- *queue*<*base*>::**queue_head** — Returns a pointer to the first element of the queue.
- *queue*<*base*>::**queue_single** — Returns true if the queue has one and only one element.
- *queue*<*base*>::**queue_tail** — Returns a pointer to the last element of the queue.
- *queue*<*base*>::**remqueue** — Removes a specified element of the queue.
- *queue*<*base*>::**replace** — Removes a specified element, placing a new one in its place in the list.
- *queue*<*base*>::**try_remqueue** — Looks to see if an element is in the queue, removing it if so.

A slight variant of *queue* is *counted_queue*<*base*>, with the same methods as *queue*<*base*>, but with the inclusion of a count of the number of members.

- *counted_queue*<*base*>::**get_count** — Returns a count of the number of queue elements.

Also defined is a pair of template classes, *iterated_queue*<*base*> and *iterated_queue_element*<*base*>, with the same methods as *queue*<*base*> and *queue_element*<*base*>, but with the addition of an iteration point that is properly maintained as the queue is otherwise manipulated. The iteration point supports logic that iterates over all of the elements in the queue.

- *iterated_queue*<*base*>::**iterate_init** — Reset the iteration point to the beginning of the queue and return the first element.
- *iterated_queue*<*base*>::**iterate** — Returns the element after that previously returned by **iterate** or **iterate_init**.

Kernel time values are represented by the *kernel_time* structure, for which the arithmetic and comparison operators are defined.

Security Policy

This library has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

These library routines use no special privileges or capabilities and have no dependence on other subsystems.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

These library routines are protection critical because they:

- Are expected by other TCB components to function correctly.

Functional Decomposition

This subsystem consists of the functions listed in the subsystem decomposition.

Concurrency Issues

All of the routines in this subsystem require that the caller assume responsibility for correct multi-threaded operation when applied to common data.

Languages Employed

Basic data structure classes are implemented in C++. The ANSI C library functions are implemented in C or assembly language as appropriate for performance.

APPENDIX A

Primitive Bootstrap Server Design

Overview

The Primitive Bootstrap Server is the first user task started by the MK++ kernel during system initialization. It is responsible for starting the rest of the servers needed to support system users. The basic functionality described below is common to both trusted and untrusted configurations; it is anticipated that this subsystem will serve as the basis for the enhancements necessary to initialize the TCB servers.

Server Interface

The Primitive Bootstrap Server receives its start-up controls from three sources:

- Options entered at the system console in response to the `boot:` prompt.
- Default options compiled into the definition of the object.

Once the Primitive Bootstrap Server has completed initialization of the other servers, it provides default service for the *bootstrap* and *exception* ports of its "children".

System Console Boot Options

In response to the `boot:` prompt, the console operator is allowed to specify the disk file name of the MK++ kernel executable/Primitive Bootstrap Server, and to override the default bootstrap configuration file. The name of desired configuration file may be provided directly or the **-a** argument may be used to cause the Primitive Bootstrap Server to prompt the operator for the file later in the initialization process. **-q** is accepted as a synonym for **-a**.

Default Configuration

By default, one instance of the OSF/1 server will be loaded from /servers/osf1_server and started. No arguments are passed.

Configuration File

Each entry in /servers/bootstrap.conf (or the file provided in response to the -a boot argument, above) describes a server to be started. Entries are delimited by new-line characters and contain one or more fields that are delimited by spaces; the fields after the first are not position-dependent.

- *server_name*: The symbol table name of the server.
- *server_path*: The pathname of the file, which defaults to /servers/server_name.
- *server_args*: A continuous string, starting with a hyphen *or* a multi-part string delimited by < and >.
- *security_id*: Two comma-separated hexadecimal fields, enclosed in *{}* braces. If omitted, sequential values will be assigned.
- *port_mask*: Two hexadecimal digits in the form 0xff that identify the ports to be provided to **bootstrap_ports**. If omitted, a default mask will be applied.
- Any additional fields will be passed as arguments to the newly started server.

Bootstrap Port Handling

The Primitive Bootstrap Server services the *bootstrap* ports of the tasks it spawns. The following requests are supported:

- **bootstrap_arguments** — Returns task arguments.
- **bootstrap_ports** — Returns the privileged ports permitted for that server task.

Exception Port Handling

The Primitive Bootstrap Task also provides default *exception* ports for the tasks it spawns. All exceptions are fatal to the spawned task.

Security Policy

This server has no special role in the enforcement of system security other than enforcing per-server port distribution masks. The servers it starts are responsible for establishing the trusted system environment.

Privileges and Capabilities

This server holds all of the system privileged ports, which it obtains via a **bootstrap_ports** call upon its task bootstrap port. The server distributes these port rights to the tasks it starts. In a trusted environment, only the TCB servers are started by the Primitive Bootstrap Server. In turn, only trusted servers will start any untrusted user tasks and will grant or deny access to the privileged ports according to system security policy.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This server is protection critical because it:

- Is expected by other TCB components to function correctly.

Initialization Sequence and Dependencies

The Primitive Bootstrap Server is the first and only user task started directly by the kernel initialization code. By the time it is given control, the kernel is fully operational but lacks a default memory manager to handle pageable memory. The Primitive Bootstrap Server is responsible for starting the Default Memory Manager and some system server, as a minimum. It may be configured to start additional servers, as is the case for trusted configurations.

Functional Decomposition

The Primitive Bootstrap Server is built as a static instance of the *bootstrap_server* object class. It uses an array of task_entry objects to describe the tasks it starts. This array is initialized from the bootstrap configuration file, if provided, or from a default_config array that is statically defined. Each task is started in order and the procedure then goes into server mode to handle the *bootstrap* and *exception* ports of its children.

Concurrency Issues

This server is single threaded.

Servers and Libraries Used

Because of the unique role this server plays, and given the fact that this server has no role in the enforcement of the system security policy, this server does not use the class libraries designed for use in TCB servers. The only library used is a subset of the ANSI C library.

Languages Employed

This server is implemented in C.

APPENDIX B

Default Memory Manager Server Design

Overview

Although it executes in user mode (as a thread within the bootstrap task), the default memory manager may be logically considered a part of the kernel because it provides anonymous pageable memory for user tasks. The kernel itself does not page its data structures, so any access to anonymous memory so provided is made on behalf of a user request.

Most of the management of anonymous virtual memory resides in the kernel. The default memory manager simply provides the mechanism to move pages back and forth between memory and the paging store. The default memory manager presents the abstraction of a single backing space object, whose contents are addressed by kernel request to store and retrieve anonymous pages.

Server Interface

The kernel engages in an asynchronous dialogue with the default memory manager in as much as that the kernel itself never waits for a response (but user tasks do wait). This dialogue is a subset of the external memory management protocol specified in the *Kernel Interface*s document. The only messages the kernel will ever send to the default memory manager are **memory_object_data_return** and **memory_object_data_request**; the only responses the kernel will accept from the manager are **memory_object_data_supply** and **memory_object_data_error**. The manager itself exports only one service interface, **add_partition**, which is called by appropriate trusted entities to supply the manager with the device partitions to be used as backing space.

Security Policy

This server has no role in contributing to the enforcement of the system security policy.

Privileges and Capabilities

This server uses the system virtual paging object control port to satisfy kernel requests for anonymous pages.

Use of Protection Mechanisms

No special TCB protection mechanisms are used.

Protection Criticality

This server is protection critical because it:

- Provides system facilities used to represent objects and subjects (memory).
- Provides system resources which are used to implement objects and subjects (memory).
- Is expected by other TCB components to function correctly.

Initialization Sequence and Dependencies

Until this server is operating, no other task can access pageable anonymous memory. The Primitive Bootstrap task is completely wired to ensure that none of its memory is paged out. Until the default memory manager supplies an abstract backing space port, the kernel will evict no anonymous pages.

When the Primitive Bootstrap task creates and starts the default memory manager (a thread within the bootstrap task), the default memory manager initializes its data structures and announces its presence to the kernel, via **vm_set_default_memory_manager**. The manager also creates a service port which the bootstrap task can provide to other system tasks. These other tasks can request and use this service port to provide the device ports of backing space for the manager to use. Until some device ports are supplied, any pages evicted to the default memory manager will reside in memory.

Functional Decomposition

To maximize I/O performance, the default memory manager endeavors to perform multi-page I/O operations whenever possible. Whenever the kernel makes a page-in or page-out request of the manager (which can be multi-page), the manager tries to keep the requests intact, so that each kernel request can translate into a single I/O request. There are a variety of circumstances in which the manager cannot do this, basically when:

- A kernel request spans multiple disk partitions.

- Some of the requested pages are in transit between memory and backing store.

To deal with these problems, the manager maintains separate lists (all indexed by virtual paging file index) of backing store partitions, pages in memory, pages in transit and pages requested. As additional requests come from the kernel, those requests are split into sub-requests on the basis of these lists, and the lists updated accordingly.

Concurrency Issues

The default memory manager is single-threaded. All operations are performed asynchronously, though, so tracking the state of operations is necessary, and is performed as part of request splitting described above.

Servers and Libraries Used

Because of the unique role this server plays, and given the fact that this server has no role in the enforcement of the system security policy, this server does not use the class libraries designed for use in TCB servers. The only library used is a subset of the ANSI C library.

This server depends on the Primitive Bootstrap server for its connection to the rest of the trusted servers.

Languages Employed

This server is implemented in C++.