Mach 3 Kernel Principles

Open Software Foundation and Carnegie Mellon University

Keith Loepere

This book is in the **Open Software Foundation Mach 3** series.

Books in the OSF Mach 3 series:

Mach 3 Kernel Principles

Mach 3 Kernel Interfaces

Mach 3 Server Writer’s Guide

Mach 3 Server Writer’s Interfaces

Revision History:

Revision 2	MK67: January 7, 1992	OSF Mach release
Revision 2.2	NORMA-MK12: July 15, 1992	

Change bars indicate changes since MK67.

Contents

CHAPTER 1 Introduction

This book documents the user visible architecture of the Mach 3 kernel. It is assumed that the reader is familiar with the basic ideas of Mach as are found in:

> Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young, “Mach: A New Kernel Foundation for UNIX Development,” in *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA.

The notion of operating system functionality provided via a Mach user space server can be found in:

> David Golub, Randall Dean, Alessandro Forin, Richard Rashid, “UNIX as an Application Program,” in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA.

Mach was originally developed as a variant of BSD that provided users with enhanced memory management, multiple points of control (*threads*) and an extensive process (*task*) to process communication facility (IPC). The goals of Mach include:

- Exploiting parallelism in both operating system and user applications.
- Supporting large, potentially sparse address spaces with flexible memory sharing.
- Allowing transparent network resource access.
- Compatibility with existing software environments (BSD).
- Portability.

The Mach 3 kernel is a kernel that provides only the Mach related features, along with any features needed to provide support for higher system layers. As such, the kernel no longer provides BSD functionality; indeed, it does not provide the functionality of any traditional operating system. Instead, it provides the base upon which operating systems can be built. The Mach kernel subscribes to the philosophy of:

- A simple, extensible communication kernel.
- An object basis with communication channels as object references.
- A client / server programming model, using synchronous and asynchronous inter-process communication.
- User mode tasks performing many traditional operating system functions (e.g. file system, network access).

The fundamental idea is that of a simple, extensible communication kernel. It is a goal of the Mach project to move more and more functionality out of the kernel, until everything is done by user mode tasks communicating via the kernel. Of course, even in the extreme, the kernel must provide other support besides task to task communication, in particular:

- Management of points of control (*threads*).
- Resource assignment (*tasks*).
- Support of address spaces for tasks.
- Management of physical resources (physical memory, processors, device channels).

Even here, though, the goal is to move functionality outside the kernel. User mode tasks implement the policies regarding resource usage; the kernel simply provides mechanisms to enforce those policies.

Kernel Abstractions

Although it is a goal of the Mach kernel to minimize abstractions provided by the kernel, it is not a goal to be minimal in the semantics associated with those abstractions. As such, each of the abstractions provided has a rich set of semantics associated with it, and a complex set of interactions with the other abstractions. Although this makes it difficult to identify key ideas, the main kernel abstractions are considered to be the following:

- Task — The unit of resource allocation: large address space, port rights.
- Thread — The unit of CPU utilization, lightweight (low overhead).
- Port — Communication channel, accessible only via send / receive capabilities (rights).
- Message — Typed collection of data objects.
- Memory object — Internal unit of memory management.

The kernel provides some memory management, of course. Memory is associated with tasks. Memory objects are the means by which tasks take control over memory management.

Tasks and Threads

The Mach kernel does not provide the traditional notion of the *process*. This is for two main reasons:

- Any given operating system environment has considerable semantics associated with a process (such as user ID, signal state, etc.). It is not the purpose of the Mach kernel to understand or provide these extended semantics.
- Many systems (BSD, for example) equate a process with an execution point of control. Some systems (AOS, for example) do not. Mach wishes to support multiple points of control in a way separate from any given operating system environment's notion of process.

Instead, Mach provides two notions: the *task* and the *thread*. A thread is Mach's notion of the point of control. A task exists to provide resources for its containing threads. This split is made to provide for parallelism and resource sharing.

A thread:

- Is a point of control flow in a task.
- Has access to all of the elements of the containing task.
- Potentially executes in parallel with other threads, even threads within the same task.
- Has minimal state for low overhead.

A task:

- Is a collection of system resources. These resources, with the exception of the address space, are referenced by ports. These resources may be shared with other tasks if rights to the ports are so distributed.
- Provides a large, potentially sparse address space, referenced by machine address. Portions of this space may be shared through inheritance or external memory management.
- Contains some number of threads.

Note that a task has no life of its own; only threads execute instructions. When it is said "a task Y does X" what is really meant is that "a thread contained within task Y does X".

A task is a fairly expensive entity. It exists to be a collection of resources. All of the threads in a task share everything. Two tasks share nothing without explicit action (although the action is often simple) and some resources cannot be shared between two tasks at all (such as port receive rights).

A thread is a fairly light-weight entity. It is fairly cheap to create and has low overhead to operate. This is true because a thread has little state (mostly its register state); its owning task bears the burden of resource management. On a multiprocessor it is possible for multiple threads in a task to execute in parallel. Even when parallelism is not the goal, multiple threads have an advantage in that each thread can use a synchronous programming style, instead of attempting asynchronous programming with a single thread attempting to provide multiple services.

Memory Management

The Mach kernel provides the mechanisms to support large, potentially sparse virtual address spaces. Each task has an associated address map (maintained by the kernel) which controls the translation of virtual address in the task's address space into physical addresses. As is true in virtual memory systems, the contents of the entire address space of any given task is most likely not completely resident in physical memory at any given time, and mechanisms must exist to use physical memory as a cache for the virtual address spaces of tasks. Unlike traditional virtual memory designs, the Mach kernel does not implement all of this caching itself; it endeavors to allow user mode tasks the ability to participate in these mechanisms.

Unlike all other resources in the Mach system, virtual memory is not referenced via ports. Memory can be referenced only by using virtual addresses as indices into a particular task's address space. The memory (and the associated address map) that defines a task's address space can be partially shared with other tasks.

A task can allocate new ranges of memory within its address space, de-allocate them, and change protections on them. It can also specify *inheritance* properties for the ranges. A new task is created by specifying an existing task as a base from which to construct the address space for the new task. The inheritance attribute of each range of the memory of the existing task determines whether the new task has that range defined and whether that range is virtually copied or shared with the existing task.

Within Mach, most virtual copy operations for memory are actually achieved through copy-on-write optimizations. A copy-on-write optimization is accomplished by not directly copying the range, but by protected sharing. The two tasks both share the memory to be copied, but with read-only access. When either task attempts to modify a portion of the range, that portion is copied at that time. This lazy evaluation of memory copies is an important performance optimization performed by the Mach kernel, and important to the communication / memory philosophy of Mach.

Any given region of memory is *backed* by a *memory object*. A *memory manager* task provides the policy governing the relationship between the image of a set of pages while cached in memory (the physical memory contents of a memory region) and the image of that set of pages when not so cached (the abstract *memory object*). The Mach kernel comes with a default memory manager that provides basic non-persistent memory objects that are zero filled initially and paged against system paging space.

Task to Task Communication

Communication between tasks is a very important element of the Mach philosophy. Mach believes in a a client / server system structure in which tasks (clients) access services by making requests of other tasks (servers) via messages sent over a communication channel. Since the Mach kernel provides very few services of its own (in particular, it provides no file service), a Mach task will need to communicate with a potentially great many other tasks that do provide these services. These communication channels in Mach are called *ports*. A port is a unidirectional channel consisting of a (fixed length) queue that holds *messages*. A message is a typed collection of data. A port is named by port *right*s held by tasks. A task can manipulate a port only if it holds the appropriate port rights. Only one task can hold the *receive right* for a port. This one task is allowed to receive (read) messages from the port queue. Multiple tasks can hold *send rights* to the port that allow them to send (write) messages into the queue. A task communicates with another task by building a data structure that contains a set of typed data elements, and then performing a message-send operation on a port for which it holds send rights. At some later time, the task with receive rights to that port will perform a message-receive operation. Note that this message transfer is an asynchronous operation. The message is logically copied into the receiving task (possibly with copy-on-write optimizations). Multiple threads within the receiving task can be attempting to receive messages from a given port, but only one thread will receive any given message.

The Mach kernel does not understand distribution at all (unless configured with the experimental multicomputer support, which provides distributed shared memory and IPC within a collection of Mach nodes). However, the Mach IPC facility is designed so that a server task (the Net Message server) can transparently forward messages over a network.

Structure of this Document

The next chapter of this document provides a brief, non-formal model of the system architecture supported by the Mach kernel. This attempts to discuss each feature of the kernel in an isolated way, building to an understanding of the programming model supported, without suggesting any particular method of use.

The remaining chapters of this document discuss each feature in turn, so as to allow programmers to better understand how to use these features in a reasonably consistent manner.

CHAPTER 2 Architectural Model

Like all systems, the Mach system has, as its primary responsibility, the provision of points of control that execute instructions within some framework. In Mach, these points of control are called *threads*. Threads execute in a *virtual* environment. The virtual environment.provided by the Mach kernel consists of a *virtual* processor that executes all of the user space accessible hardware instructions, augmented by emulated instructions (system traps) provided by the kernel; the virtual processor accesses a set of (virtualized) registers and some *virtual* memory that otherwise responds as does the machine's physical memory; all other hardware resources are accessible only via special combinations of memory accesses and emulated instructions. Note that all resources provided by the Mach kernel are virtualized. This chapter describes, at top level, the elements of the virtual environment seen by Mach threads.

Elements

The Mach kernel provides an environment consisting of the following elements:

- thread — An execution point of control. A thread is a light-weight entity; most of the state pertinent to a thread is associated with its containing task.
- task — A container to hold references to resources in the form of a port name space, a virtual address space and a set of threads.
- port — A unidirectional communication channel between tasks.
- port set — A set of ports which can be treated as a single unit when receiving a message.
- port right —A capability allowing particular rights to access a port.
- port name space — An indexed collection of port names each of which names a particular port right.

- message — A typed collection of data passed between two tasks.
- message queue — A queue of messages associated with a single port.
- virtual address space — A sparsely populated indexed set of memory pages that may be referenced by the threads within a task. Ranges of pages may have arbitrary attributes and semantics associated with them via mechanisms implemented by the kernel and external memory managers.
- abstract memory object — An abstract object that represents the non-resident state of the memory ranges backed by this object. The task that implements this object is called a memory manager.
- memory cache object — A kernel object that contains the resident state of the memory ranges backed by an abstract memory object.
- processor — A physical processor capable of executing threads
- processor set — A set of processors, each of which can be used to execute the threads assigned to the processor set.
- node — An individual multiprocessor within a multicomputer.
- host — The multiprocessor/multicomputer as a whole.
- device — Physical device accessible by user mode tasks.
- event — A kernel device maintained signalling event count.

Each of these notions will be discussed in detail. However, since some of their definitions depend on the definitions of others, some of the key notions will be discussed in simplified form so that a full discussion can be understood.

A *thread* is the basic computational entity. A thread belongs to one and only one *task* that defines its virtual address space. To affect the structure of the address space, or to reference any resource other than the address space, the thread must execute a special trap instruction which causes the kernel to perform operations on behalf of the thread, or to send a message to some agent on behalf of the thread. In general, these traps manipulate resources associated with the task containing the thread. The resources provided by the Mach kernel that can be directly manipulated are:

- threads
- tasks (and associated virtual address spaces and port name spaces)
- processors and processor sets
- hosts (and, in some cases, nodes)
- devices and events

These entities are resources in the sense that the kernel provides them, and requests can be made of the kernel to manipulate these entities: to create them, delete them and affect their state.

The kernel is just one manager that can provide resources (those listed above) and provide services. Tasks may also provide services, and implement abstract resources themselves. The kernel provides communications methods that allow a client task to request that a server task (actually, a thread executing within it) provide a service. In this way, a task has a dual identity; one identity is that of a resource managed by the kernel, whose resource manager executes within the kernel; the second is that of a supplier of resources for whom the resource manager is the task itself.

With the exception of the task's virtual address space, all other system resources are accessed through a level of indirection known as a *port*. A port is a unidirectional communication channel between a client who requests a service and a server who provides the service. (If a reply is to be provided to such a service request, a second port must be used.) The service to be provided is determined by the manager that receives the message sent over the port. It follows that the receiver for ports associated with kernel provided entities is the kernel and the receiver for ports associated with task provided entities is the task providing that entity. For ports that name task provided entities, it is possible to change the receiver of messages for that port to be a different task. A single task may have multiple ports that refer to resources it supports. For that matter, any given entity can have multiple ports that represent it, each implying different sets of permissible operations. For example, many entities have a *name* port and a *control* port (sometimes called the privileged port). Access to the control port allows the entity to be manipulated; access to the name port simply names the entity, for example, to return statistics.

There is no system-wide name space for ports. A thread can access only the ports known to its containing task. A task holds a set of *port rights*, each of which names a (not necessarily distinct) port and which specifies the rights permitted for that port. Port rights can be transmitted in messages; this is how a task gets port rights. A port right is named with a port name, which is an integer chosen by the kernel that is meaningful only within the context (port name space) of the task holding that right.

Most operations in the system consist of sending a message to a port that names some manager for the object being manipulated. In this document, this will be shown in the form:

object → **function**

which means that the **function** is to be invoked (by sending an appropriate message) to a port that names the *object*. Since a message must be sent to some port (right), this operation has an object basis. However, not all entities are named by ports and so this is not a pure object model. The two main non-port right named entities are port names/rights themselves, and ranges of memory. (*Event* objects are also named by task local IDs.) To manipulate a memory range, a message is sent to the containing virtual address space (named by the owning task). To manipulate a port name/right (and, often, the associated port), a message is sent to the containing port name space (named by the owning task). A subscript notation,

object [*id*] → **function**

is here used to show that an *id* is required as a parameter in the message to indicate which range or element of *object* is to be manipulated. This form is also used for a handful of operations in which a privileged port (the host control port) must be supplied as well as the object to be manipulated for the sake of verifying privilege for the operation.

Threads

A *thread* is the basic computational entity. A thread belongs to one and only one *task* that defines its virtual address space. A thread has the following state:

- Its machine state (registers and the like), which change as the thread executes and which can also be changed by a holder of the kernel thread port.
- A (small) set of thread specific port rights, identifying the thread's kernel port and a port used to send exception messages on behalf of the thread.
- A suspend count, non-zero if the thread is not to execute instructions.
- Resource (scheduling) parameters.
- Various statistics, including statistical PC samples.

A thread operates by executing instructions in the usual way. Various special instructions trap to the kernel, to perform operations on behalf of the thread. The most important of these kernel traps is the **mach_msg_trap**, which allows the thread to send messages to kernel and other servers to operate upon resources for it. (This trap is almost never directly called; it is invoked via the **mach_msg** library routine.)

Exceptional conditions arising during the thread's execution (floating point overflow, page not resident, etc.) are handled by sending messages to some port. The port used depends on the nature of the condition. The outcome of the exceptional condition depends on setting the thread's state and/or responding to the exception message.

The operations that can be performed upon a thread are:

- Creation and destruction.
- Suspension and resumption (manipulating the suspend count).
- Machine state manipulation.
- Special port manipulation.
- Resource (scheduling) control.
- Statistical PC sampling.

Tasks

A task can be viewed as a container that holds a set of threads. It contains default values to be applied to its containing threads. Most importantly, it contains those elements that its containing threads need to execute, namely, a port name space and a virtual address space. The state associated with a task is:

- The set of contained threads.
- The associated virtual address space.
- The associated port name space, naming a set of port rights, and a related set of port notification requests.

- A (small) set of task specific ports, identifying the task's kernel port, a default port to use for exception handling for contained threads and bootstrap ports to name other services.
- Emulation library addresses for routines that gain control upon the attempted execution of certain system call instructions.
- A suspend count, non-zero if no contained threads are to execute instructions.
- Default scheduling parameters for threads.
- Various statistics, including statistical PC samples.

Tasks are created by specifying a prototype task which specifies the host on which the new task is created, and which can supply (by inheritance) various portions of its address space.

The operations that can be performed upon a task are:

- Creation and destruction.
- Suspension and resumption.
- Special port manipulation.
- Manipulation of contained threads.
- Statistical PC sampling of the contained threads.

Ports

A port is a unidirectional communication channel between a client who requests a service and a server who provides the service. A port has a single receiver and (potentially) multiple senders.

The major state associated with a port is its associated message queue. A port also maintains a count of references (rights) to it.

Kernel services exist to allocate ports. Every system entity (other than virtual memory ranges) is named by a port, so ports are also created implicitly when these entities are created.

The kernel will provide notification messages upon the death of a port upon request.

Messages

A message is a typed collection of data passed between two entities. A message is not a system object in its own right. However, since messages are queued, they are significant because they can hold state between the time a message is sent and when it is received. This state consists not only of pure data; it also consists of virtual copy memory ranges and port rights.

Message Queues

A port basically consists of a queue of messages. This queue is manipulated only via message operations (**mach_msg**) that transmit messages. The state associated with a queue is the ordered set of messages queued, and a settable limit on the number of messages.

Port Rights

A port can only be accessed via a port *right*. A port right is an entity that indicates the right to access a specific port in a specific way. In this context, there are three types of port rights:

- receive right — Allows the holder to receive messages from the associated port.
- send right — Allows the holder to send messages to the associated port.
- send-once right — Allows the holder to send a single message to the associated port. The right self-destructs after the message is sent.

Port rights can be copied and moved between tasks via various options in the **mach_msg** call, and also by explicit command. Other than message operations, port rights can be manipulated only as members of a port name space.

Port rights are created implicitly when any other system entity is created and explicitly via explicit port creation.

The kernel will, upon request, provide notification (to a port of one's choosing) when there are no more send rights to a port. The destruction of a send-once right (other than by using it to send a message) generates a send-once notification sent to the corresponding port.

Port Name Space

Ports and port rights do not have system-wide names that allow arbitrary ports or rights to be manipulated directly. Ports can be manipulated only via port rights, and port rights can be manipulated only when they are contained within a port *name space*. A port right is specified by a port *name* which is an index into a port name space. Each task has associated with it a single port name space.

An entry in a port name space can have four possible values:

- MACH_PORT_NULL — No associated port right.
- MACH_PORT_DEAD — A right was associated with this name, but the port to which the right referred has been destroyed.
- a port right — A send-once, send or receive right for a port.
- a port set name — A name which acts like a receive right, but that allows receiving from multiple ports.

Acquiring a new right in a task generates a new port name. As port rights are manipulated (by referring to their port names) the port names are sometimes themselves manipulated. All send and receive rights to a given port in a given port name space will have the same port name. Each send-once right to a given port will have a different port name from each other and from the port name used for any send or receive rights held.

Operations supported for port names include:

- Creation (via creation of a right) and deletion.
- Query of the associated type.
- Rename.

The kernel will provide notification of a name becoming unusable upon request.

Since port name spaces are bound to tasks, they are created and destroyed with their owning task.

Port Sets

A port set is a set of ports which can be treated as a single unit when receiving a message. A **mach_msg** receive operation is allowed against a port name that either names a receive right, or a port set. A port set contains a collection of receive rights. When a receive operation is performed against a port set, a message will be received from one of the ports in the set. The received message will indicate from which member port it was received. It is not allowed to directly receive a message from a port that is a member of a port set. There is no notion of priority for the ports in a port set; there is no control provided over the kernel's choice of the port within the port set from which any given message is received.

Operations supported for port sets include:

- Creation and deletion.
- Membership changes and membership queries.

Virtual Address Spaces

A virtual address space defines the set of valid virtual addresses that a thread executing within the task owning the virtual address space is allowed to reference. A virtual address space is named by its owning task.

A virtual address space consists of a sparsely populated indexed set of pages. The attributes of individual pages may be set at will. For efficiency, the kernel groups virtually contiguous sets of pages that have the same attributes into internal memory regions. The kernel is free to split or merge memory regions at will. System mechanisms are sensitive to the identities of memory regions, but most user accesses are not so affected, and can span memory regions freely.

A given memory range can have distinct semantics associated with it through the actions of a memory manager. When a new memory range is established in a virtual address space, an abstract memory object is specified (possibly by default) that represents the semantics of the memory range, by being associated with a task (a memory manager) that provides those semantics.

A virtual address space is created when a task is created, and destroyed when the task is destroyed. The initial contents of the address space is determined from various options to the **task_create** call, as well as the inheritance properties of the memory ranges of the prototype task used in that call.

Most operations upon a virtual address space name a memory range within the address space. These operations include:

- Creating (allocating) and de-allocating a range.
- Copying a range.
- Setting special attributes, including “wiring” the page into physical memory to prevent eviction.
- Setting memory protection attributes.
- Setting inheritance properties.
- Directly reading and writing ranges.

Abstract Memory Objects

The Mach kernel allows user mode tasks to provide the semantics associated with referencing portions of a virtual address space. It does this by allowing the specification of an abstract *memory object* that represents the non-resident state of the memory ranges backed by this memory object. The task that implements this memory object (that responds to messages sent to the port that names the memory object) is called a *memory manager.*

The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various memory objects. The kernel is involved in an asynchronous dialog with the various memory managers to maintain this cache, filling and flushing this cache as the kernel sees fit, by sending messages to the abstract memory object ports.

The operations upon abstract memory objects include:

- Initialization.
- Page reads.
- Page writes.
- Requests for permission to access pages.
- Page copies.
- Termination.

Memory Cache Objects

The portion of the kernel’s main memory cache that contains the resident pages associated with a given abstract memory object is referred to as the *memory cache* object. The memory manager for a memory object holds send rights to the kernel’s memory cache object. The memory manager is involved in an asynchronous dialog with the kernel to provide the abstraction of its abstract memory object by sending messages to the associated memory cache object.

The operations upon memory cache objects include:

- Set attributes (including initialized state).
- Return attributes.
- Supply pages to the kernel.
- Indicate that pages requested by the kernel are not available.
- Indicate that pages requested by the kernel should be filled by the kernel’s default rules.
- Restrict access to memory pages.
- Termination.

Processors

Each physical processor (that is capable of executing threads) is named by a processor control port. Although significant in that they perform the real work, processors are not very significant in the Mach scheme of things other than as members of a processor set. It is a processor set that forms the basis for the pool of processors used to schedule a set of threads, and that has scheduling attributes associated with it.

The operations supported for processors include:

- Assignment to a processor set.
- Machine control, such as start and stop.

Processor Sets

Processors are grouped into processor sets. A processor set forms a pool of processors used to schedule the threads assigned to that processor set. A processor set exists as a basis to uniformly control the schedulability of a set of threads. The notion also provides a way to perform coarse allocation of processors to given activities in the system.

The operations supported upon processor sets include:

- Creation and deletion.
- Assignment of processors.
- Assignment of threads and tasks.
- Scheduling control.

Nodes

In general, the Mach kernel executes on a single machine, possibly a multiprocessor. Multiple such machines may be connected together in various ways, but this is the province of user space tasks. However, optional (and experimental) support is provided within the kernel for *multicomputers*, “machines” consisting of multiple multiprocessors (without shared memory between multiprocessors). Each uniprocessor or multiprocessor in such a multicomputer is called a *node* and referenced by a node ID (a number). The multicomputer as a whole is a Mach *host*.

Mach’s multicomputer support provides transparently distributed shared memory between nodes and transparently distributed Mach IPC between nodes. The only direct operations supported by nodes are the setting and retrieving of a small set of node specific ports.

Hosts

Each machine (uni-processor or multi-processor) in a networked Mach system runs its own instantiation of the Mach kernel. The *host* multiprocessor is not generally manipulated by client tasks. But, since each host does carry its own Mach kernel, each with its own port space, physical memory and other resources, the executing host is visible and sometimes manipulated directly. Also, each host generates its own statistics.

Hosts are named by a name port which is freely distributed and which can be used to obtain information about the host and a control port which is closely held and which can be used to manipulate the host. Operations supported by hosts include:

- Clock manipulation.
- Statistics gathering.
- Re-boot.
- Set the default memory manager.
- Obtain lists of processors and processor sets.

Devices

The Mach kernel exports a very simple interface to its devices. When initialized, the Mach kernel builds an internal table that lists each device. It exports a single port, the *device master* port, which is responsible for allocating devices. A task that holds send rights to the device master port may request the kernel to *open* a device, returning a port that provides access to that device. Operations on that port then manipulate the device, until it is *closed*.

Operations on devices include:

- Read and write.
- Status return and setting.

- Special purpose operations.
- Mapping a shared memory window between a user space task and the device / device driver.

Events

In support of user space device drivers, kernel device drivers may provide *event* objects that count significant events of a certain type and signal (make running) a thread waiting for such an event. The set of events is chosen by the device driver (possibly as a result of instructions given to the driver); there is no general (explicit) way to create or delete event objects. Event objects are named by an event ID local to the accessing task, as chosen by the driver and communicated to the task in a driver specific way. The only supported operation for event objects is to wait for an event to occur. The event object maintains a count of outstanding events so that events are not lost.

Note that event objects and their interface are very preliminary and highly subject to change.

CHAPTER 3 Threads and Tasks

Threads are the active entities in a Mach system. They act as points of control within a task, which provides them with a virtual address space and a port name space with which other resources are accessed.

This chapter discusses the user visible view of threads and tasks.

It is common practice to discuss scheduling issues when discussing threads. However, the issue of scheduling is only marginally related to the semantics of threads. Issues directly involved in the actual scheduling of threads on processors (processor sets and the like) are discussed in the chapter on physical resources.

Threads

A *thread* is the basic computational entity. A thread belongs to one and only one *task* that defines its virtual address space. A thread is a light-weight entity with a minimum of state. A thread executes in the way dictated by the hardware; fetching instructions from its task's address space based on the thread's register values. The only actions a thread can take directly are to execute instructions that manipulate its registers and read and write into its memory space. An attempt to execute privileged machine instructions, though, causes an exception (discussed later). To affect the structure of the address space, or to reference any resource other than the address space, the thread must execute a special trap instruction which causes the kernel to perform operations on behalf of the thread, or to send a message to some agent on behalf of the thread. Also, faults or other illegal instruction behavior cause the kernel to invoke its exception processing.

FIGURE 1 shows the client visible structure associated with a thread. The thread object is the receiver for messages sent to the kernel thread port. Aside from any random task that may hold send rights to this thread port, the thread port is also accessible via the thread’s thread self port, the containing task or the containing processor set.

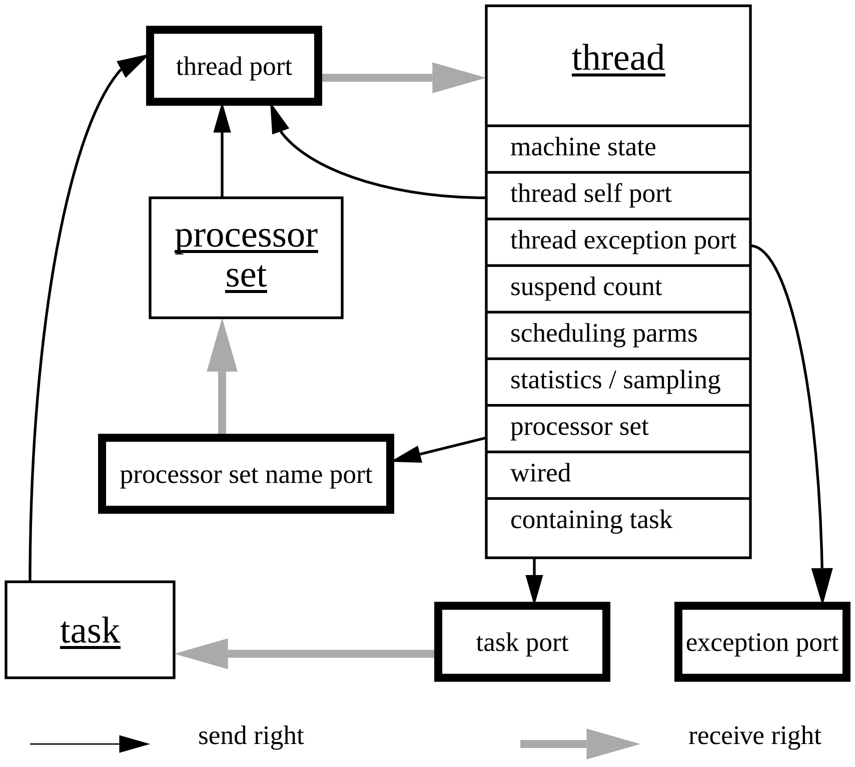

FIGURE 1 **Thread Structures**

Actions by Threads

This section describes the details of the actions that a thread can take directly. Of course, a thread can do anything if it can gain access to the correct port rights and send messages to them; the various things it can do are discussed under the sections describing the object manipulated.

Scheduling Support Traps

The Mach kernel preemptively schedules threads. The way in which this is done is related to various factors. For now, it is sufficient to say that threads have scheduling priority associated with them which is used to select which threads should execute within a given processor set (discussed as part of physical resources).

thread_switch causes a context switch with various options. It is provided for cases (such as software lock routines) that want to give up the processor so that other threads can make progress. The options mostly have to do with selecting the appropriate new thread to run, when this information is available.

One of the options to **thread_switch** causes the scheduling priority of the thread to be depressed to the lowest possible value so that other threads will run, thereby (it is hoped) completing work that blocks this depressed thread. This priority depression is canceled when the given time expires, the thread is next run regardless of the depression, *thread* →**thread_abort** is called or *thread* →**thread_depress_abort** is called.

Two additional traps, **swtch,** which attempts to context switch to a different thread, and **swtch_pri**, which switches and also sets the current thread's scheduling priority, are being phased out in favor of **thread_switch**.

The **evc_wait** trap causes the invoking thread to wait for a kernel (device) defined event. This is described along with **device_map**. (Note that this feature is subject to change.)

Identity Traps

Other than the few traps mentioned in this chapter, all other requests for services require a port right. Even requests upon the kernel that manipulate the current thread or task need a port right (naming the current thread or task). To bootstrap this process, a thread needs a way, without any port right, to get the port right for itself and its task. These rights are obtained through the **mach_thread_self** and **mach_task_self** traps, respectively.

The port rights returned are actually the THREAD_KERNEL_PORT and TASK_KERNEL_PORT special ports last set through the *thread* → **thread_set_special_port** and *task* → **task_set_special_port** message calls. The default values for these special ports are the actual kernel thread and task ports, respectively. The creator of a task or thread can set these special port values before starting the thread or task so that the thread or task does not have access to its own kernel ports, but instead invokes some intermediate port when requesting services to be done to itself.

The kernel also provides a trap, **mach_host_self**, which return a (send) right to the host's name port.

Bootstrap Reply Port Trap

The **mach_reply_port** trap is also used for bootstrap purposes. As mentioned earlier, if a service request is to return a reply, a second port is needed. This trap is used to create an initial reply port (a receive right) that can then be used for all other port related calls.

Message Send and Receive Trap

The final, and most important trap, is **mach_msg_trap** (the trap invoked by the **mach_msg** library routine). This is the trap that provides access to all other system services. It sends and/or receives a message to/from a port named by a given right. The semantics of this call are very involved, and described in detail in the *Kernel Interface* document, and also in various chapters in this document.

Exception Processing

A thread has an exception port associated with it. When an exception occurs in a thread (these exceptions are listed under **catch_exception_raise** in the *Kernel Interface* document), the thread, executing in kernel context, sends a message whose contents describe the exception to its exception port. A successful reply to this message causes the thread to continue (in a state possibly altered by **thread_set_state**). If the reply message has a non-success return value, or the thread's exception port is not defined, the kernel will try sending the exception message to the task's exception port. As before, a successful reply causes the thread to continue. If this message receives a non-success reply, or the task's exception port is not defined, the thread is terminated.

Not every exceptional condition that a thread encounters is handled in this way.

For example, a page not resident fault does not send a message to the exception port; instead, a message is sent to the external memory manager associated with the memory page in which the faulting address lies (this is discussed as part of virtual memory).

The general exception rule does not apply to the system call instruction(s). First of all, several of the possible system call numbers are subsumed for Mach kernel calls. The remainder are initially undefined. An attempt to execute them results in an exception as defined above. At the granularity of a task, an indirection can be supplied separately for each system call number. This is done via *task* → **task_set_emulation** or *task* → **task_set_emulation_vector** (and examined with *task* → **task_get_emulation_vector**). When a thread attempts to execute a system call whose number has been redirected, the system call is effectively translated into a type of subroutine call to the address specified with the **task_set_emulation** call. This allows a task to establish an emulation library within its address space to be used by threads emulating an existing operating system.

Actions on Threads

This section lists the various things that can be done to a thread, given a send right to the kernel's thread port.

Life and Death

A thread is created via *task* → **thread_create** and destroyed via *thread* → **thread_terminate**. Since a thread belongs to a given task, thread creation is actually an operation performed upon a task. The result is a send right to the kernel's thread port for the new thread. A list of the kernel thread ports for all of the threads in a given task can be obtained with *task* → **task_threads**.

A newly created thread is in the *suspended* state. This is the same as if *thread* → **thread_suspend** had been called upon it prior to its executing its first instruction. A suspended thread does not execute. A thread is created in the suspended state so that its machine state can be properly set before it is started. To remove a thread from the suspended state (actually, to decrement its suspend count), *thread* → **thread_resume** is used.

Thread State

A thread has two main sets of state, its machine state and a set of special ports.

The machine state for a thread is obtained via *thread* → **thread_get_state** and set via *thread* → **thread_set_state**.

The result of setting a thread's state at a random point is undefined. Various steps are needed to obtain a deterministic result.

- *thread* → **thread_suspend** is used to stop the thread. This, and the following step, are unnecessary if the thread has just been created and has yet to run. They are needed, though, for exception processing or for asynchronous interruption (such as signal delivery).
- *thread* → **thread_abort** is called. This causes any system call (really, **mach_msg** or any related message call, such as exception or page missing messages) to be aborted. Aborting a message call sets the thread's state to be at the point after the system call, with a return value indicating interruption of the call. Aborting a page fault or exception leaves the thread at the point of the page fault or exception; resuming the thread will cause it to retake the page fault or exception.
- *thread* → **thread_set_state** can then be safely used. (Note: Some hardware allows page faults or exceptions to occur mid-instruction. This mid-instruction state may or may not be permitted to be manipulated.)
- *thread* → **thread_resume** restarts the thread.

A thread has two special ports associated with it. The first is the value for the thread to use to request operations upon itself. This is normally the same as the kernel thread port, but can be different if so set (most likely by the creator of the thread). The second is the port used by the thread, within the kernel, when it is processing exceptions on behalf of itself. These two ports are returned by *thread* → **thread_get_special_port** and set by *thread* → **thread_set_special_port**.

Various random pieces of kernel thread state, such as the suspend count and scheduling information, can be obtained via *thread* → **thread_info**.

Scheduling Control

The following functions affect the scheduling of a thread. They are described under processor sets.

- *thread* → **thread_assign**
- *thread* → **thread_assign_default**
- *thread* → **thread_get_assignment**
- *thread* → **thread_max_priority**
- *thread* → **thread_policy**
- *thread* → **thread_priority**
- *host_control* (*thread*)→ **thread_wire**

The **thread_wire** call marks the thread as "wired"—privileged with respect to kernel resource management. A "wired" thread is always eligible to be scheduled and can consume memory even when free memory is scarce. This property is assigned to threads within the default page-out path. Threads not in the default page-out path should not have this property to prevent the kernel's free list of pages from being exhausted.

task (*thread*) → **mach_sample_thread** provides support for the Posix **profil** system call. (The call requires both a thread parameter and a task parameter on the same host.) After being called, the kernel will start sampling the program counter (PC) of the specified thread periodically (whenever it is running when a clock interrupt occurs). Buffers of these PC values will be sent when full to a port specified in the sample call.

Tasks

A task can be viewed as a container that holds a set of threads. It contains default values to be applied to its containing threads. Most importantly, it contains those elements that its containing threads need to execute, namely, a port name space and a virtual address space.

FIGURE 2 shows the client visible task structures. The task object is the receiver for messages sent to the kernel task port. Aside from any random task that may hold send rights to this task port, the task port is also accessible via the task's task self port, the contained threads or the containing processor set.

Life and Death

A new task is created with *task* → **task_create**. Note that task creation is an operation requested of an existing ("prototype") task. The result is a new task located on the same machine as the prototype task (not that of the task making this invocation). The new task can either be created with an empty virtual address space, or one inherited from the prototype task. The new task's port name space is empty. The new task inherits the parent task's PC sampling state.

A task is destroyed with *task* → **task_terminate**. This operation is requested of the task to be destroyed, not the parent specified in its creation. The task's virtual address space and port name space are destroyed.

Various random statistics about the task can be obtained with *task* → **task_info**.

Special Ports

Aside from its associated port name space, a task also has a small set of special ports. These are the so-called "special" ports and the "registered" ports.

A task has three "special" ports associated with it. The first is the value for the task to use to request operations upon itself. This is normally the same as the kernel task port, but can be different if so set (most likely by the creator of the task). The second is the port used by a thread, within the kernel, when it is processing exceptions on behalf of itself (when it has not set a thread specific exception port). The third is a bootstrap port, which can be used for anything, but which is intended as the initial port a task holds to something other than itself, for use in locating other services. These ports are returned by *task* → **task_get_special_port** and set by *task* → **task_set_special_port**. The value of these ports in a new task are inherited from the task that was the target of the **task_create** call (with the obvious exception of the task self port).

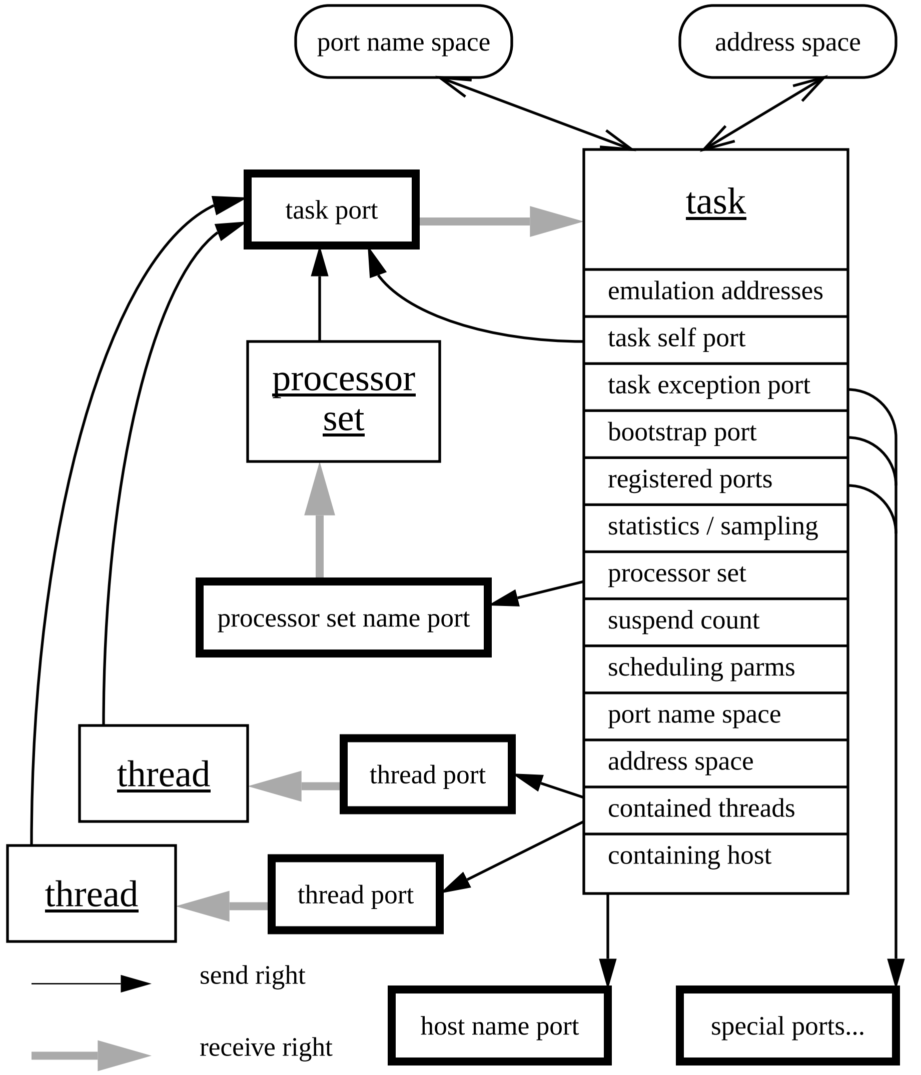

FIGURE 2 Task Structures

A task also has a small array of "registered" ports, which also inherit from the task used in the **task_create** call. These ports can be set with *task* → **mach_ports_register** and returned by *task* → **mach_ports_lookup**. Although these ports can have any use, their expected uses are to refer to the Network Name server, the Environment server and the "Service" server.

Thread Management

A thread belongs to one and only one task. Threads are created with *task* → **thread_create**. The set of threads present in a task can be found with *task* → **task_threads**.

Although a task does not itself execute, some execution properties can be set for a task which will then apply to its contained threads. All of the threads in a task can be suspended or resumed en masse by *task* → **task_suspend** and *task* → **task_resume**. These operations do not affect the threads' suspend counts; they affect the task's suspend count. A thread can execute only if both its and its task's suspend counts are zero.

As mentioned under exception processing for threads, a thread that executes a system call instruction can have that system call redirected back into user space. This is accomplished by setting an emulation routine address (for the task as a whole) with *task* → **task_set_emulation** or *task* → **task_set_emulation_vector**. The emulation vector can be examined with *task* → **task_get_emulation_vector**.

The default scheduling properties for threads can be set with the following:

- *task* → **task_assign**
- *task* → **task_assign_default**
- *task* → **task_get_assignment**
- *task* → **task_priority**

task(*task*) → **mach_sample_task** provides support for the Posix **profil** system call. (The call requires both the task port and a reference task on the same host.) After being called, the kernel will start sampling the program counter (PC) of all threads within the task periodically (whenever one is running when a clock interrupt occurs). Buffers of these PC values will be sent when full to a port specified in the sample call. Since a single port is specified for this call, all PC samples for all threads will be randomly mixed in the buffers sent to that port.

CHAPTER 4 Ports, Rights and Messages

With the exception of its shared memory, a Mach task interacts with its environment purely by sending messages and (hopefully) receiving replies. These messages are sent via ports, communication channels with multiple senders and single receivers. A task holds rights to these ports that specify its ability to send or receive messages.

Ports

A port is a unidirectional communication channel between a client who requests a service and a server who provides the service.

A port has a single receiver and (potentially) multiple senders. A port that represents a kernel supported resource has the kernel as the receiver; this receivership cannot change. A port that names a service provided by a task has that task as the port's receiver; this receivership can change if desired, as discussed under port rights.

The major state associated with a port is its associated message queue. A port also maintains a count of references (rights) to it.

FIGURE 3 shows a typical port, illustrating a series of extant send rights and the single receive right. The associated message queue has a series of ordered messages. One of the messages is shown in detail, showing its destination port, reply port reference, a send and a receive right being passed in the message, as well as some out-of-line (virtual copy) memory.

Few operations affect the port itself; most operations affect port rights or a port name space containing those rights, or affect the message queue.

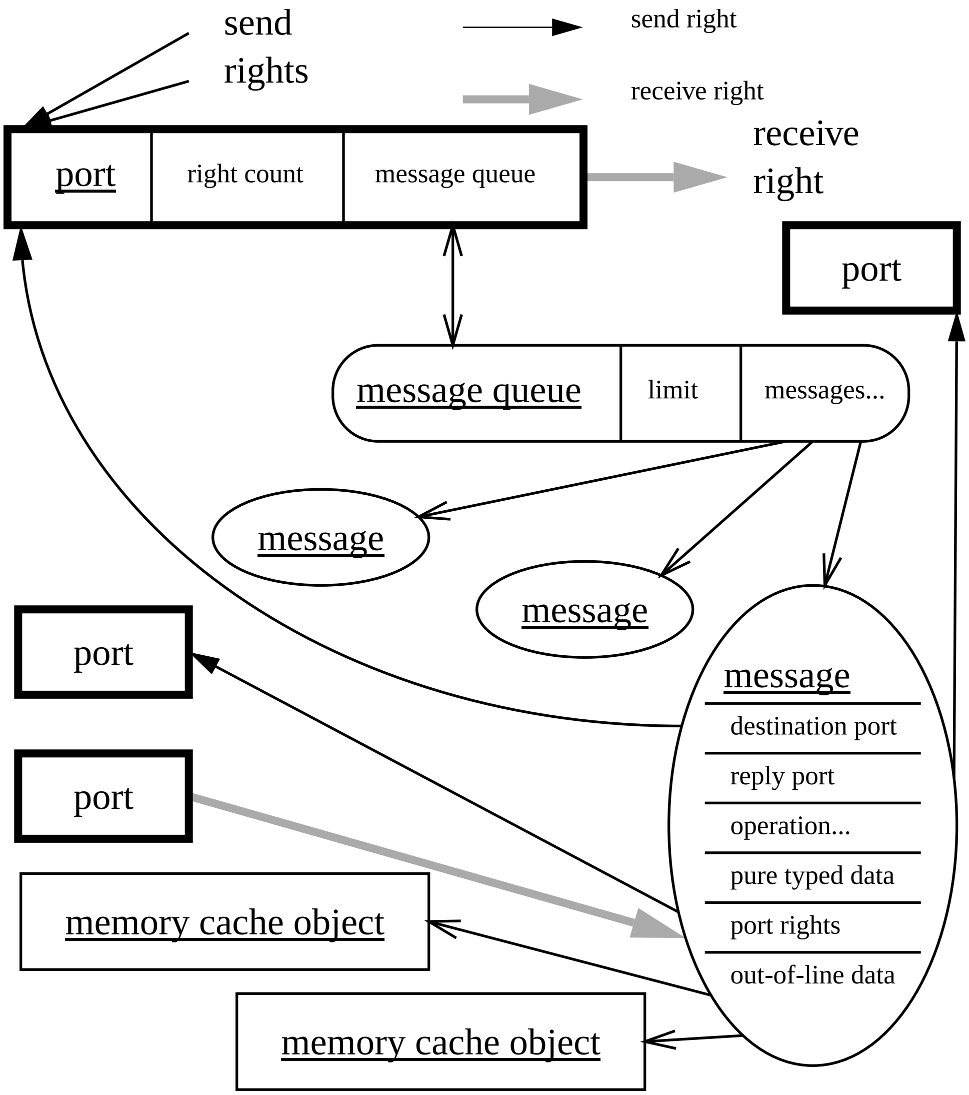

FIGURE 3 **Port Structures**

Ports are created implicitly when any other system entity (threads, tasks, processors, processor sets, hosts or devices) is created. Also, **mach_reply_port** creates a port. Ports are created explicitly by *port_name_space* → **mach_port_allocate** and *port_name_space* [*port_name*]→ **mach_port_allocate_name**. A port cannot be explicitly destroyed. It is destroyed only when the receive right is destroyed.

The existence of ports is of obvious importance to all involved. As such, many tasks using a port may well wish to be notified, through a message, when they die. Such notifications are requested with options to **mach_msg**, as well as with *port_name_space* [*port_name*]→ **mach_port_request_notification**. The way in which this destruction becomes evident depends on the viewer (and is dictated by the method in which the notification was requested). The possibilities are:

- dead name notification — A port has been destroyed. The message indicates the task's name for the now dead port. (This is discussed under port name spaces.) A notification of this form can be requested with *port_name_space* [*port_name*]→ **mach_port_request_notification** or with the MACH_RCV_NOTIFY option to **mach_msg**.
- port destroyed notification — A port was to be destroyed. The message carries the receive right, thereby saving the port. A notification of this form can be requested (*port_name_space* [*port_name*]→ **mach_port_request_notification**), given the receive right. The movement (to another task) of the receive right does not affect any existing port destroyed notification requests. (This feature is currently planned to be deleted from the kernel interface.)

Messages

A message is a typed collection of data passed between two entities. A message is not a manipulable system object in its own right. However, since messages are queued, they are significant because they can hold state between the time a message is sent and the time when it is received.

Besides pure data, a message can also contain port rights. This is very significant. It is in this way (in general) that a task obtains new rights, by receiving them in a message. The ways in which this is possible are too numerous to discuss here; refer to the description of **mach_msg** in the *Kernel Interface* document.

A message consists of a fixed sized header (**mach_msg_header_t**) followed by the data items contained in the message. The header specifies a port name for the port to which the message is sent, a port name for the port to which a reply is to be sent (if a reply is requested), the message size and operation code fields.

The data items follow the header. Each consists of a data descriptor (**mach_msg_type_t** or **mach_msg_type_long_t**) followed by the data. The type descriptor specifies the type of the data, as well as a count of the number of data items of this type.

A message can contain references to "out-of-line" memory, as indicated by its data descriptors. Like the other parts of the message, it is virtually copied from the sender to the receiver. The kernel uses significant copy-on-write virtual memory optimizations to make the passing of large data efficient. For our-of-line data, the data descriptor is followed by the virtual address of the data, instead of the data itself. When a message is received that contains out-of-line memory, this memory will appear as newly allocated memory (as if by **vm_map**), using the same memory manager as the sender's memory used.

Each type descriptor includes an optional *deallocate* flag, which is meaningful only for port rights and out-of-line memory. (The use of the appropriate IPC port types that specific the disposition of ports is preferred to the use of the deallocate flag for port rights.) If set, the act of queueing the message will de-allocate the port rights and/or memory range from the sending task.

Message Queues

A port basically consists of a queue of messages. This queue is manipulated only via message operations (**mach_msg**) that transmit messages.

The only controllable state for a message queue is its size. This can be set with *port_name_space* [*port_name*] → **mach_port_set_qlimit** given the receive right for the associated port. If a message queue is full, no more messages can be queued (callers will block). However, **mach_msg** provides an option allowing one message to be left waiting to be queued. In this case, when the queue is no longer full, the message is then queued, and a notification is sent.

Messages sent to a port are delivered reliably (messages may not be lost) and are received in the order in which they were sent by any given thread.

Port Rights

A port can only be accessed via a port *right*. A port right is an entity that indicates the right to access a specific port in a specific way. In this context, their are three types of port rights:

- receive right — Allows the holder to receive messages from the associated port.
- send right — Allows the holder to send messages to the associated port.
- send-once right — Allows the holder to send a single message to the associated port. The right self-destructs after the message is sent.

Port rights are a secure, location independent way of identifying ports. These rights are kernel protected entities; clients manipulate port rights only via port names they have to these rights.

mach_msg is one of the principal ways that rights are manipulated. Port rights can be moved between tasks (deleted from the sender and added to the receiver) in messages. Also, option flags in a message will cause **mach_msg** to make a copy of an existing send right, or to generate a send or a send-once right from a receive right. Rights can also be forcefully copied or moved by *port_name_space* [*port_name*]→ **mach_port_extract_right** (the equivalent of the target sending the right in a message) and *port_name_space* [*port_name*]→ **mach_port_insert_right** (the equivalent of the target receiving the right in a message).

Other than message operations, port rights can be manipulated only as members of a port name space.

FIGURE 4 shows a series of port rights, some contained in a port name space and some in transit in a message. Also shown in the port name space is a port set.

Port rights are created implicitly when any other system entity is created. Also, **mach_reply_port** creates a port right. Port rights are created explicitly by *port_name_space* → **mach_port_allocate** and *port_name_space* [*port_name*]→ **mach_port_allocate_name**.

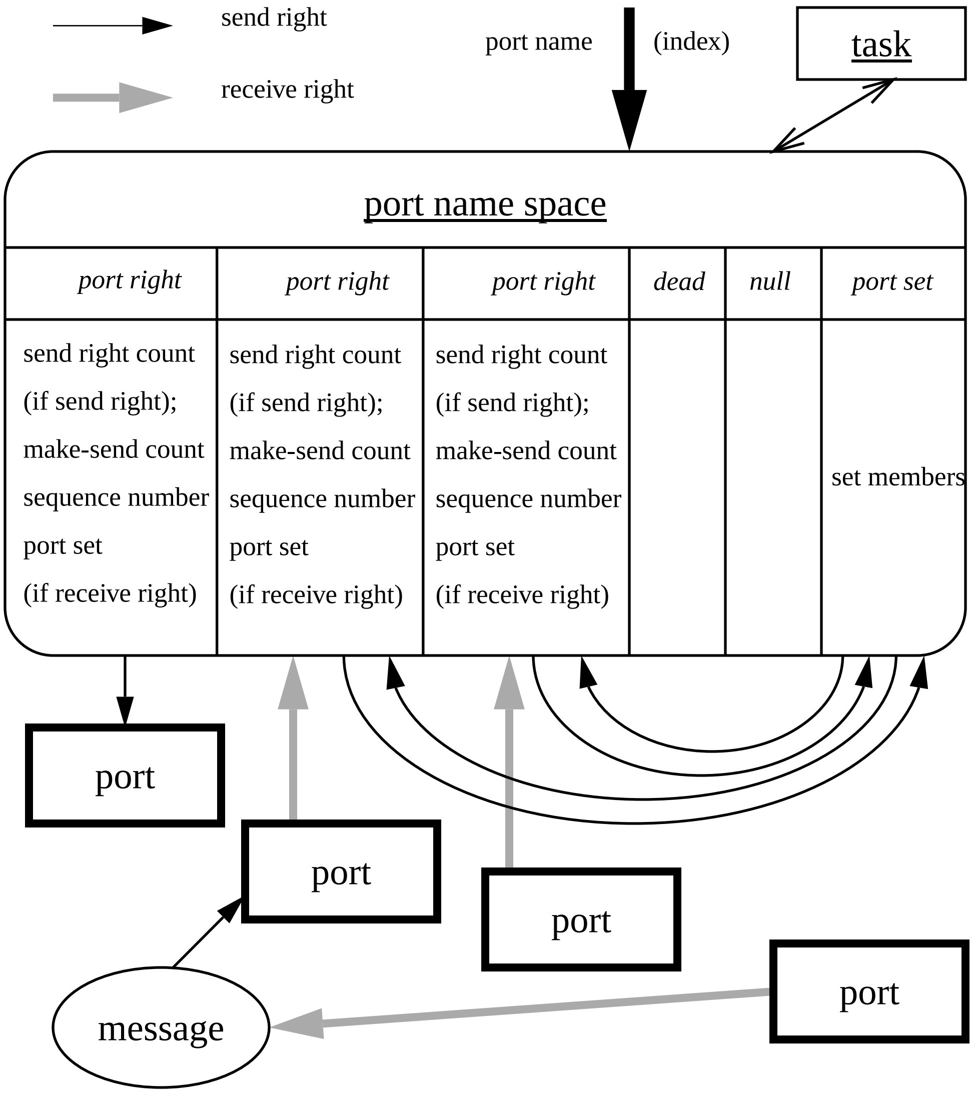

FIGURE 4 **Port Right Structures**

A port right is destroyed by *port_name_space* [*port_name*]→ **mach_port_deallocate** and *port_name_space* [*port_name*]→ **mach_port_destroy**. Destruction can also be a by-product of port name space manipulations, such as by *port_name_space* [*port_name*]→ **mach_port_mod_refs**.

Some status information can be obtained, given a receive right, with *port_name_space* [*port_name*]→ **mach_port_get_receive_status**.

The system maintains a (system-wide) count of the number of send (and send-once) rights for each port (this includes rights in transit in messages, including the destination and reply port rights). The receiver of a port may well be interested if there are no more send rights for the port, indicating that the port may no longer have value. A notification of this form can be requested (*port_name_space* [*port_name*]→ **mach_port_request_notification**). This notification depends on the notion of a make-send count, discussed as a part of port name spaces. The movement (to another task) of the receive right does not currently affect any existing no-more-senders notification requests. (A planned change is to cancel outstanding no-more-senders notification requests, and to send a send-once notification to indicate this cancelation.)

A send-once right allows a single message to be sent via it. These rights are generated only from the receive right. A send-once right has the property that it guarantees that a message will result from it. In the normal case, a send-once right is consumed by using it as the destination port in a message; the right is (silently) destroyed when the message is received. The send-once right can be moved from task to task (other than being used as a destination right) until such time as it is consumed. If the right is destroyed in any way other than by using it to send a message, a send-once notification is sent to the port instead.

Most of the ways in which a send-once right can be destroyed (other than by using it to send a message) are fairly obvious. There are several obscure cases:

- The send-once right was specified as the target for a no-senders notification and the port for which the no-senders notification was requested is deleted. Since there will be no forth coming no-senders notification, a send-once notification is generated instead.
- The send-once right was specified as the target for a message-accepted notification and the port for which the message was waiting to be accepted is deleted. No message-accepted notification is generated in this case; a send-once notification is generated instead.
- In the process of performing a non-atomic message receive, the task gives away its receive right after the message is de-queued from the port but prior to its being returned to the task. A send-once notification is sent to the destination port signifying the lost association between the message sent via the send-once right and the port.

The complete rules for manipulation of port rights are too complicated to be described here; refer to the *Kernel Interface* document.

Port Name Space

Ports, themselves, are not named. It is the port rights that are. A port right can only be named by being contained within a port *name space*. A port is specified by a port *name* which is an index into a port name space. Each task has associated with it a port name space.

An entry in a port name space can have four possible values:

- MACH_PORT_NULL — No associated port right.
- MACH_PORT_DEAD — A right was associated with this name, but the port to which the right referred is now dead. The port name is kept in this state until explicit

action is taken to avoid reusing this name before the client task understands what happened.

- a port right — A send-once, send or receive right for a port.
- a port set name — A name which acts like a receive right, but that allows receiving from multiple ports. This is discussed in the next section.

Each distinct right that a port name space contains does not necessarily have a distinct name in the port name space. Send-once rights always consume a separate name for each distinct right. Receive and send rights, though, to the same port coalesce. That is, if a port name space holds three send rights for some port, it will have a single name for all three rights. A port name has an associated reference count for each type of right (send-once, send, receive, port set and dead name) associated with the name. If the port name space also holds the receive right, that receive right will have the same name as the send right.

A name becomes dead when its associated port is destroyed. (It follows that a task holding a dead name cannot be holding a receive right under that name as well.) The dead name only has a non-zero reference count for the number of send or send-once references previously held by that name. A task can be notified (a message sent to it) when one of its names becomes dead via *port_name_space* [*port_name*]→ **mach_port_request_notification**. Receiving this notification message increments the reference count for the dead name, to avoid a race with any threads manipulating the name.

Whenever a task acquires a right (by whatever means) it is assigned a port name subject to the above rules. Acquiring a right increments the name’s reference count for the type of that right. The reference count can be obtained with *port_name_space* [*port_name*]→ **mach_port_get_refs**.

Although a port name can be explicitly destroyed (*port_name_space* [*port_name*]→ **mach_port_destroy**) thereby removing all references, port names are typically manipulated by modifying the user reference count. *port_name_space* [*port_name*]→ **mach_port_mod_refs** modifies the reference count for a specified right type associated with a name. *port_name_space* [*port_name*]→ **mach_port_deallocate** is similar to **mach_port_mod_refs**, but it always decrements the count by 1, and it will only decrement the send (or send-once) reference count. This routine is useful for manipulating the reference count for a port name that may have become dead since the decision was made to modify the name. Options to **mach_msg** that actually move a right (and also *port_name_space* [*port_name*]→ **mach_port_extract_right**) can cause the name’s reference count to be decremented. Port names are freed when all the reference counts go to zero.

If a port name is freed and a dead-name notification is in effect for the name, a port-deleted notification is generated. As such, a name with a dead-name notification in effect can be in only one of three states:

- naming a valid right
- MACH_PORT_DEAD, with a dead-name notification having been sent when the name became dead
- MACH_PORT_NULL, with a port-deleted notification having been sent when the name became null

Information about a name, in particular, the type of the name, can be obtained with *port_-name_space* [*port_name*]→ **mach_port_type**. The list of assigned names is obtained with *port_name_space* [*port_name*]→ **mach_port_names**. The name by which a right is known can be changed with *port_name_space* [*port_name*]→ **mach_port_rename**.

Some status information can be obtained, given a receive right name, with *port_name_space* [*port_name*]→ **mach_port_get_receive_status**.

Port names that name receive rights have an associated make-send count, used for no-more-sender notification processing. The make-send count is the kernel's count of the number of times a send right was made from the receive right (with a message element that is a port right specifying the MACH_MSG_TYPE_MAKE_SEND type descriptor for **mach_msg**). This make-send count is set to zero when a port is created, and reset to zero whenever the receive right is transmitted in a message. It can also be changed with *port_name_space* [*port_name*]→ **mach_port_set_mscount**. The make-send count is included in the no-more-senders notification message. Note that a no-senders notification indicates the lack of extant send rights; there may still be outstanding send-once rights. A task can easily keep track of the send-once rights since every send-once right guarantees a message or send-once notification.

Received messages are stamped with a sequence number, taken from the port from which the message was received. (Messages received from a port set are stamped with a sequence number from the appropriate member port.) Sequence numbers placed into sent messages are overwritten. Newly created ports start with a zero sequence number, and the sequence number is reset to zero whenever the port's receive right is moved. It can also be set explicitly with *port_name_space* [*port_name*]→ **mach_port_set_seqno**. When a message is de-queued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The de-queue and increment operations are atomic, so that multiple threads receiving messages from a port can use the *msgh_seqno* field to reconstruct the original order of the messages.

Since port name spaces are bound to tasks, they are created and destroyed with their owning task.

Port Sets

A port set is a set of ports which can be treated as a single unit when receiving a message. A **mach_msg** receive operation is allowed against a port name that either names a receive right, or a port set. A port set contains a collection of receive rights. When a receive operation is performed against a port set, a message will be received at random from one of the ports in the set (the first to have a message, if only one port has a message queued). Each of the receive rights in the set has its own name, and the set has its own name. A receive against a port set reports the name of the receive right whose port provided the message. A receive right can belong to only one port set. A task may not directly receive from a receive right that is in a port set.

A port set is created with *port_name_space* [*port_name*]→ **mach_port_allocate** or *port_name_space* [*port_name*]→ **mach_port_allocate_name**. It is destroyed by *port_-*

name_space [*port_name*]→ **mach_port_destroy** or *port_name_space* [*port_name*]→ **mach_port_deallocate**.

Manipulations of port sets is done with *port_name_space* [*port_name*]→ **mach_port_move_member**. This call can add a member to a set, remove it from a set, or move it from one set to another.

The membership of a port set can be found with *port_name_space* [*port_name*]→ **mach_port_get_set_status**.

Message Transmission

The **mach_msg** system call sends and receives Mach messages.

The send operation queues a message to a port. The caller blocks until the message can be queued, unless one of the following happens:

- The message was being sent to a send-once right. These messages always forcibly queue.
- The queue was full and the caller specified the MACH_SEND_NOTIFY option. This option will force the destination port to accept a single message (returning an appropriate status), and send the caller a notification when the message is actually queued.
- The **mach_msg** operation was aborted (**thread_abort**). Note that, by default, the **mach_msg** library routine retries operations that are interrupted.
- The send operation exceeded its time-out value.
- The port was destroyed.

The message carries a copy of the caller’s data. (Data specified as out-of-line in the message is passed as a virtual copy.) After the send, the caller can freely modify the message buffer or the out-of-line memory ranges and the message contents will remain unchanged.

Aside from the obvious sending failures (invalid port rights or data formats, for example), a message may also fail to be queued because the send time-out value is exceeded or an interrupt (**thread_abort**) occurred. In these situations, the kernel tries to return the message contents to the caller with a pseudo-receive operation. This prevents the loss of port rights or memory which only exist in the message, for example, a receive right which was moved into the message, or out-of-line memory sent with the de-allocate bit.

The pseudo-receive operation is very similar to a normal receive operation. The pseudo-receive handles the port rights in the message header as if they were in the message body. After the pseudo-receive, the message is ready to be resent. If the message is not resent, note that out-of-line memory ranges may have moved and some port rights may have changed names.

The receive operation de-queues a message from a port. The receiving task acquires the port rights and out-of-line memory ranges carried in the message. The caller must supply a buffer into which the header (and any in-line data) is to be copied. If the message does not fit, it is normally destroyed. An option (MACH_RCV_LARGE), though, allows the caller to receive an error, along with the buffer size that would be needed, so that another receive operation can be attempted with an appropriate sized buffer.

A received message can contain port rights and out-of-line memory. Received port rights and memory should be consumed or de-allocated in some fashion. Resource shortages that prevent the reception of a port right or out-of-line memory destroy that entity.

The receive operation can also specify a time-out value. It may also be aborted (**thread_abort**). These situations do not affect the message that would have been received.

There are two notifications that can be requested as a result of a **mach_msg** call. The first is the msg-accepted notification, sent when a message is successfully queued after being forcibly sent with the MACH_SEND_NOTIFY option. The other notification is not generated by **mach_msg**, but is requested by the MACH_RCV_NOTIFY option. This option causes the reply port right that is received to automatically have a dead name notification requested for it (as if by **mach_port_request_notification**). This latter option is an optimization for a certain class of RPC interactions. The dead name notification on the reply port name allows the receiver of the message to be informed in a timely manner of the death of the requesting client. However, since the reply right is typically a send-once right, sending the reply will destroy the right and generate a port-deleted notification instead. An optimization to cancel this notification is provided by the MACH_SEND_CANCEL option to **mach_msg**.

Message operations are only atomic with respect to the manipulation of the port rights in message headers.

CHAPTER 5 Virtual Memory Management

Mach is well known for its virtual memory design, one that cleanly layers the virtual memory system into a machine dependent and a machine independent portion. The machine dependent portion provides a simple interface for validating, invalidating and setting the access rights for pages of virtual memory (maintaining the hardware address maps). The machine independent portion provides support for logical address maps (mapping a virtual address space), memory ranges within this map, and the interface to the backing storage (memory objects) for these ranges via the external memory management interface.

The virtual memory system is designed for uniprocessors and shared memory multi-processors of a moderate number of processors. It has been ported to non-uniform access memory architectures, although optimal support for these architectures, as well as more complex mapped hardware (such as virtually addressed caches) is still being investigated.

High performance is a feature of the Mach virtual memory design. Much of this results from its efficient support of large, sparse address spaces, shared memory, and virtual copy memory optimizations.

Finally, the virtual memory system allows clients to provide the backing storage for memory ranges, thereby defining the semantics that apply to such ranges.

Virtual Address Spaces

A virtual address space defines the set of valid virtual addresses that a thread executing within the task owning the virtual address space is allowed to reference. A virtual address space is named by its owning task.

A virtual address space consists of a sparsely populated indexed set of memory pages. The kernel internally groups virtually contiguous sets of pages that all have the same attributes (backing abstract memory object, inheritance, protection and other properties) into internal entities called memory regions. Memory regions are named only by their virtual address ranges within their containing address space. Various operations and system mechanisms are sensitive to the identities of memory regions, but most user accesses are not so affected, and can span memory regions freely. The kernel is free to split and merge memory regions as it sees fit; the client view of its address space is as a set of pages. The only call that is sensitive to memory regions is *virtual_address_space* [*memory_address*] → **vm_region**. This call locates the region "near" a given address, and returns information about that memory region. Since memory regions are purely a kernel internal notion (but affected by **vm_map** calls, as well as by changing protection and inheritance attributes), the result of this call can change from invocation to invocation.

A virtual address space is created when a task is created, and destroyed when the task is destroyed. When a new task is created (ignoring inheritance), its address space is empty and must be built through manipulations of the virtual address space before the task can have threads set into execution. An option, though, to the **task_create** call allows for the new task to "inherit" ranges of memory from the prototype task used in the create call. The kernel function *virtual_address_space* [*memory_range*] → **vm_inherit** can change the inheritance properties for a memory range, to allow or disallow inheritance. The possibilities for inheritance are:

- VM_INHERIT_NONE — The range is left undefined in the new task.
- VM_INHERIT_COPY — The range is copied (with copy optimizations) into the new task at the time of task creation.
- VM_INHERIT_SHARE — The new and old tasks share access to the memory range.

A range of memory can have distinct semantics associated with it through the actions of a memory manager. When a new memory range is established in a virtual address space, an abstract memory object is specified (possibly by default) that represents the semantics of the memory range, by being associated with a task (a memory manager) that provides those semantics. No kernel calls exist for a task to directly affect the semantics associated with its memory ranges. A task has such control only by virtue of choosing memory managers that provide the functionality desired, or by sending messages directly to its memory managers to direct their actions.

virtual_address_space [*memory_range*] → **vm_map** is the basic call to establish a new range of virtual memory. This call specifies the details of the memory range (placement, size, protection, inheritance, object offset). The significant parameter to this call is a port naming an abstract memory object that backs the storage for the range. A null port specifies that the system's default memory manager is to be used. The default memory manger provides initially zero-filled storage that is paged against the system's paging space and which will not persist between system boot-loads. *virtual_address_space* [*memory_r-ange*] → **vm_allocate** can be viewed as a simplified form of the **vm_map** call when the default memory manager is desired.

Random ranges of memory space can be made invalid with *virtual_address_space* [*memory_range*] → **vm_deallocate**.

FIGURE 5 shows the client visible virtual memory structures. Shown are three memory ranges (sets of pages), two of which have the same backing abstract memory object but possibly differing inheritance or protection attributes. One of the memory cache / abstract memory object pairs is shown in detail with the associated memory manager task.

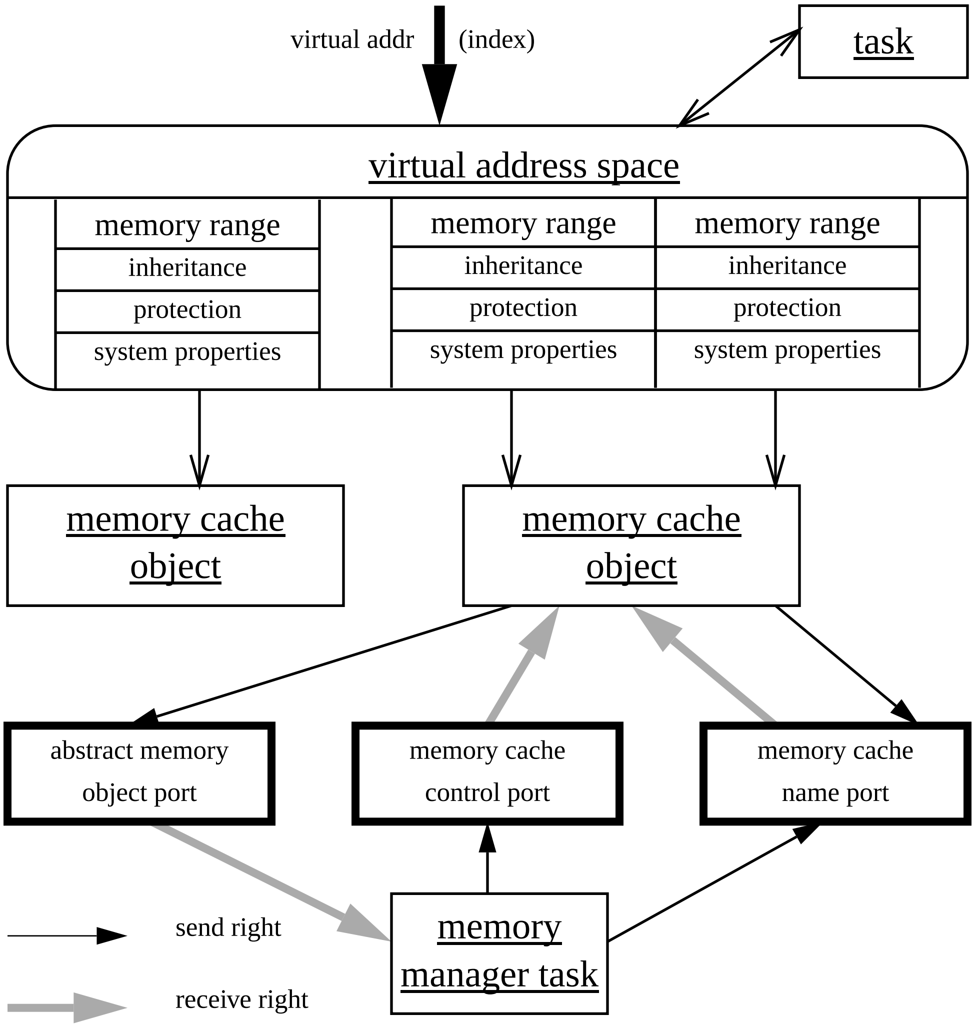

FIGURE 5 Virtual Memory Structures

Aside from the obvious hardware memory accesses allowed (as given by the range of valid addresses and the protection attributes for those ranges), the kernel also supports explicit memory manipulations. The operations supported against a span of a virtual address space are the following:

- *virtual_address_space* [*memory_range*] → **vm_copy** — Copy a memory range from one place in an virtual address space to another.
- *virtual_address_space* [*memory_range*] → **vm_machine_attribute** — Set machine specific hardware properties for the memory range.

- *virtual_address_space* [*memory_range*] → **vm_protect** — Set the allowed accesses for a memory range. Each memory range has a current and a maximum protection mask. The memory manager for a range can specify the maximum protections for all users of a range of pages; each task then has its own private maximum protection value to further restrict the allowed permissions, as well as its current protection mask.
- *virtual_address_space* [*memory_range*] → **vm_read** — Copy out a memory range.
- *host_control* (*virtual_address_space* [*memory_range*]) → **vm_wire** — "Wire" (force to be and stay resident) a range of memory. The ability to set the pageability of memory is a privileged operation and so it requires the host control port.
- *virtual_address_space* [*memory_range*] → **vm_write** — Copy in a memory range.

Memory Objects

The Mach kernel allows user mode tasks to provide the semantics associated with the act of referencing portions of a virtual address space. It does this by allowing the specification of an abstract *memory object* that represents the non-resident state of the memory ranges backed by this memory object. The task that implements this memory object (that responds to messages sent to the port that names the memory object) is called a *memory manager.*

Basic Manipulation

Manipulation of a virtual address space by a user mode task takes the following basic form:

- A task establishes a new memory range, specifying a port to name the memory object that backs that range (*virtual_address_space* [*memory_range*] → **vm_map**).
- The task attempts to reference a portion of this memory range (most likely simply by touching it). Since that portion does not yet exist in memory, the referencing task takes a page not resident fault. The kernel sends a message to the range's abstract memory object requesting the missing data. The reply from the abstract memory object resolves the requesting task's page fault.
- Eventually, the resident pages of the memory range, with values possibly modified by client tasks, are evicted from memory. Pages are sent in messages to the range's abstract memory object for their disposition.
- The client task de-establishes the memory range (*virtual_address_space* [*memory_range*] → **vm_deallocate**). When all mappings of this memory object are gone, the abstract memory object is terminated.

The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various memory objects. The portion of this cache that contains resident pages for a memory object is referred to as the *memory cache* object. The kernel is involved in a dialog with the various memory managers to maintain this cache, filling and flushing this cache as the kernel sees fit. This dialog consists, in general, of asynchronous messages, as the kernel cannot be stalled by a memory manager, and memory managers wish the maximum possible concurrency in their operations. The messages sent by the kernel are sent via routines labeled as “Server Interfaces” in the *Kernel Interface* document; messages sent by the memory managers are labelled as normal message functions.

The abstract memory object port specified in the client’s **vm_map** call names a memory manager task that implements the abstract memory object. Each abstract memory object has an associated resident kernel memory cache object that represents the cache of resident pages for that memory object. This memory cache object has an associated control port which is supplied to the memory manager so that it may control the memory cache object (mostly to respond to kernel requests on behalf of the object). The kernel also generates a name port for this memory cache object for use by client tasks to refer to the memory cache object. It is this name port that is returned by *virtual_address_space* [*memory_address*] → **vm_region**. If an abstract memory object is mapped by tasks on more than one host, there will be that many control and name ports, one for the memory cache object on each host.

Each page in the memory cache object represents some offset within the abstract memory object. This memory object offset is the object offset specified by the client in the **vm_map** call plus an integral multiple of the length of a page. Note that a client can specify an arbitrary offset to **vm_map**. Thus, a memory object may have multiple copies of its data in memory, for different offset values specified by its clients. (It is planned that this feature be removed, restricting the client’s offsets to page boundaries.)

New and Old Memory Managers

Mach provides backward compatibility for “old form” memory managers that do not support certain mechanisms that have since been added. An old form memory manager does not have support for:

- Multi-page operations
- “Precious” pages (discussed below)
- Message replies from data supply and attribute changes

Because of these differences in level of support, old memory managers use different interfaces and receive different messages from the kernel than do new memory managers. The kernel differentiates between new and old memory managers on the basis of the form of the “I’m ready” message sent to the kernel as a response to the kernel’s **memory_object_init** message (discussed below). New form memory managers reply with **memory_object_ready**; old form managers reply with **memory_object_set_attributes**.

Memory Management Initialization

When an abstract memory object is mapped on a given host for the first time, the kernel sends a message to its abstract memory object (*abstract_memory_object* → **memory_object_init**). This message informs the memory manager that the object is being mapped on a new host. The message carries the names of the kernel generated memory cache object control port and the memory object name port.

If the memory manager responds with *memory_cache_object_control* → **memory_object_ready**, this indicates to the kernel that this is a “new form” memory manager. The

memory_object_ready call also allows the specification of the “object cache” attribute and the copy strategy (discussed below). These attributes may later be changed with *memory_cache_object_control* → **memory_object_change_attributes** and examined with *memory_cache_object_control* → **memory_object_get_attributes**.

Alternately, the memory manager can respond with the old call *memory_cache_object_-control* → **memory_object_set_attributes**, which can also set the “object cache” and copy strategy attributes. This call also allows a “ready” attribute to be set; doing so has the same effect as **memory_object_ready** but indicates to the kernel that this is an “old form” memory manager.

These calls inform the kernel that the memory manager is now ready to respond to requests on behalf of this memory object.

Basic Page Manipulation

The kernel requests data for memory with *abstract_memory_object* → **memory_object_-data_request**. It will only request single pages from old form memory managers but may request multiple pages from a new form manager. (It has been proposed to add a new attribute (to **memory_object_set_attributes**) that would allow an old form manager to request the kernel to send multi-page requests.)

The memory manager supplies the requested data with *memory_cache_object_control* → **memory_object_data_supply** (new form) or *memory_cache_object_control* → **memory_object_data_provided** (old form). These calls also supply the maximum allowed accesses for the data. If the memory manager cannot supply the data because of some error, it responds with *memory_cache_object_control* → **memory_object_data_error**. This causes the kernel to cause any threads waiting for this data to take memory failure exceptions. The memory manager can alternately reply with *memory_cache_object_control* → **memory_object_data_unavailable**. In this case, the kernel supplies the missing data**,** either zeroes, or a copy of data in the case where the kernel was performing an object copy (discussed below).

When the kernel decides to flush some memory pages belonging to this memory object, the modified pages are sent as out-of-line data in a message to the memory manager via *abstract_memory_object* → **memory_object_data_return** (new form) or *abstract_memory_object* → **memory_object_data_write** (old form). In this process, the kernel converts the physical memory pages from being resident pages associated with the memory object to being “normal” pages associated with the default memory manager. In this way, if the memory manager cannot move these pages to their destination in a reasonable time, the kernel can evict these “normal” pages using the default memory manager (which will page them to paging storage) while still allowing the memory manager access to them. Normally, though, the memory manager will copy these pages somewhere (probably send them to some device or file system) and then use **vm_deallocate** to free them from its address space, as one would any out-of-line memory received.

Memory Object Termination

When all tasks remove their mappings for the memory object, the kernel informs the memory manager with *abstract_memory_object* → **memory_object_terminate** (c.f. object cacheability below). The kernel evicts all pages of the object (unless the object has a copy strategy of MEMORY_OBJECT_COPY_TEMPORARY) prior to actually terminating the memory cache object (and sending the terminate message).

Alternatively, the memory manager can call *memory_cache_object_control* → **memory_object_destroy** to explicitly shut down the memory object. All resident pages are discarded and no more activity will be allowed for the memory object. The kernel will respond with *abstract_memory_object* → **memory_object_terminate**. De-allocating the abstract memory object port also has this effect (although, obviously, no terminate message can be sent in this case). In either case, the kernel discards all pages.

One of the object attributes set when the memory object is initialized by **memory_object_ready** or **memory_object_set_attributes**, or thereafter altered by **memory_object_change_attributes** or **memory_object_set_attributes**, is the “object cache” attribute. With this attribute set, instead of terminating a memory object when all mappings are removed (thereby evicting all cache pages), the memory object is entered into a (small) kernel object cache. If some task maps the object during this time, the object stays alive (with no additional **memory_object_init** message). If no task maps the object before the object leaves the object cache, it is then terminated. If the object cache attribute is cleared while the memory object is in this un-mapped state, the memory object will be promptly terminated. Since this attribute change can have this effect, **memory_object_change_attributes** can optionally return a reply (*reply_port* → **memory_object_change_completed**) which can be used for synchronization.

Precious Pages

In the basic data management protocol, the kernel only returns to the memory manager pages that have been modified while in the physical memory cache. When evicted, pure (un-modified) pages are discarded on the assumption that the memory manager holds a copy. For the example of a mapped file memory manager that uses disk (files) as backing store and for which space is always allocated on backing storage for each page, this is the most reasonable approach. However, for managers that use virtual memory as backing store (as does the network shared memory server or other specialized servers), this is inefficient; both the manager and the kernel hold the page while it is in use. By specifying a page as “precious” when supplying it to the kernel, the memory manager need not hold a copy; the kernel commits to returning the page when evicted. This ability to specify pages as precious is made available to new form memory managers as an option to the **memory_object_data_supply** call. The memory manager can be informed of the acceptance of these pages by requesting a reply from the data supply call. This reply, *reply_to* → **memory_object_supply_completed** indicates which pages were accepted. The reply will follow all data return messages resulting from rejected supplied pages. The **memory_object_data_return** messages will return both modified pages as well as precious pages (with a flag indicating the case).

Direct Cache Manipulation

Sometimes the memory manager needs to force flush pages, or to alter protections or the like. This is done with the *memory_cache_object_control* → **memory_object_lock_request** call. This call informs the kernel of the operations to be performed. When completed, the kernel replies with *reply_port* → **memory_object_lock_completed** (using a reply port named in the **memory_object_lock_request** call).

The main purpose of **memory_object_lock_request** is to support memory management protocols that involve setting and revoking access to pages (such as distributed memory servers, or transaction protected memory). If a thread attempts to access a page in a way not currently permitted (as established by the permissions set with **memory_object_data_provided** or **memory_object_data_supply**), the kernel sends a message to the memory manager with *abstract_memory_object* → **memory_object_data_unlock**. If the memory manager honors this request, it responds with **memory_object_lock_request** to specify the new allowed permissions. With a single client, the memory manager can refuse the request by having the kernel flush the range (with **memory_object_lock_request**) and then, when the kernel re-fetches the data, the memory manager can respond with **memory_object_data_error**.

A **memory_object_lock_request** sequence can also be started on the memory manager's initiative. The call can require the kernel to return all modified (or precious) pages to the manager (most likely also requiring the kernel to revoke write permission) when the manager needs the most recent copy of the pages. The memory manager can also require the kernel to flush the pages altogether from its memory cache.

Summary of Page Manipulations

The various page manipulations can be summarized in the following set of figures.

A page starts in the **e** state (empty—not in memory). The only allowed actions on the page in this state are for a thread to attempt an access of the page to read ("peek") or to write ("poke"). Either action causes the kernel to request the page. The kernel will make no other request of the manager until the manager responds to the data request message. Once in memory, there are many transitions that can be caused by both the client and the manager. The page may be evicted from any state (unless the page is wired).

Once a page is in memory, there are five state variables that determine its disposition.

- If the page is *wired*, it will not be evicted. Since eviction is only kernel-initiated, the presence of the *wired* bit can be viewed as if the kernel was not inclined to evict the page anyway and so is not reflected in the state diagrams.
- *Precious* pages affect the process of cleaning and evicting pages. This state does not affect the transitions, only the actions taken by the transitions and so is not reflected in the transition diagrams.
- The page may be modified (m).
- The page permits some type of access (n (none), r (read) or w (read/write)).
- The page wants more access than it currently has.

Aside from the page-in states, the various states are labeled with a triple <*modified, current access, desired access*>.

FIGURE 6 shows the various states and transitions associated with the page-in path. The transitions are as follows:

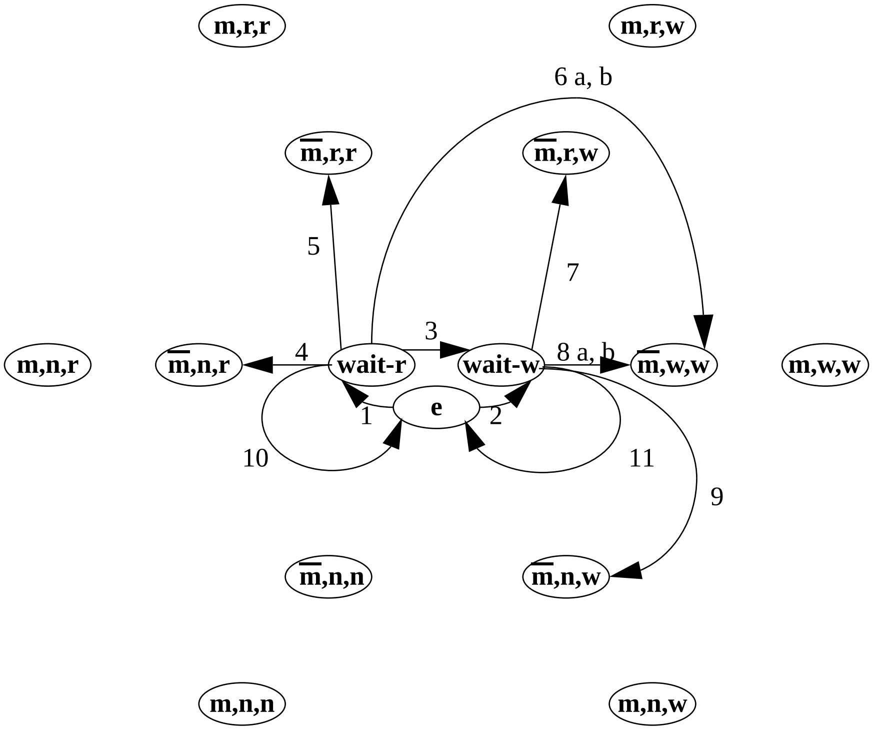

FIGURE 6 **Page-in Transitions**

- 1—Some thread “peeks” at the page. The kernel sends a **memory_object_data_request** (read access) message to the manager.
- 2—Alternately, some thread “pokes” at the page. The kernel sends a **memory_object_data_request** (write access) message to the manager.
- 3—After requesting a page with read access but before receiving one, some thread “pokes” the page. The kernel does not send a new data request message, nor an unlock request at this time. The state of the page when the page is supplied, though, will be affected (as to what access is desired).
- 4—The manager supplies the page (**memory_object_data_supply** or **memory_object_data_provided**) specifying no access. The kernel will send a **memory_object_data_unlock** message requesting read access.
- 5—The manager supplies the page with read access. The kernel simply accepts it.
- 6a—The manager responds with **memory_object_data_unavailable**. The kernel generates a writable page.
- 6b—The manager supplies the page with write access. The kernel accepts the page, marking it as writable.

- 7—The manager supplies the page with read access. The kernel will send a **memory_object_data_unlock** message requesting write access.
- 8a—The manager responds with **memory_object_data_unavailable**. The kernel generates a writable page.
- 8b—The manager supplies the page with write access. The kernel simply accepts it.
- 9—The manager supplies the page with no access. The kernel will send a **memory_object_data_unlock** message requesting write access.
- 10 and 11—The manager responds with **memory_object_data_error**. All threads waiting for the page take memory access exceptions.

The following figures show the state transitions for *lock* (change allowed accesses) and/or *clean* (return clean page copies) operations. (*Flush* operations always transition to state **e**.)

FIGURE 7 shows the manager initiated *lock* and/or *clean* transitions (**memory_object_lock_request**) when the page is in an un-modified state. FIGURE 8 shows the corresponding transitions for initially modified states.

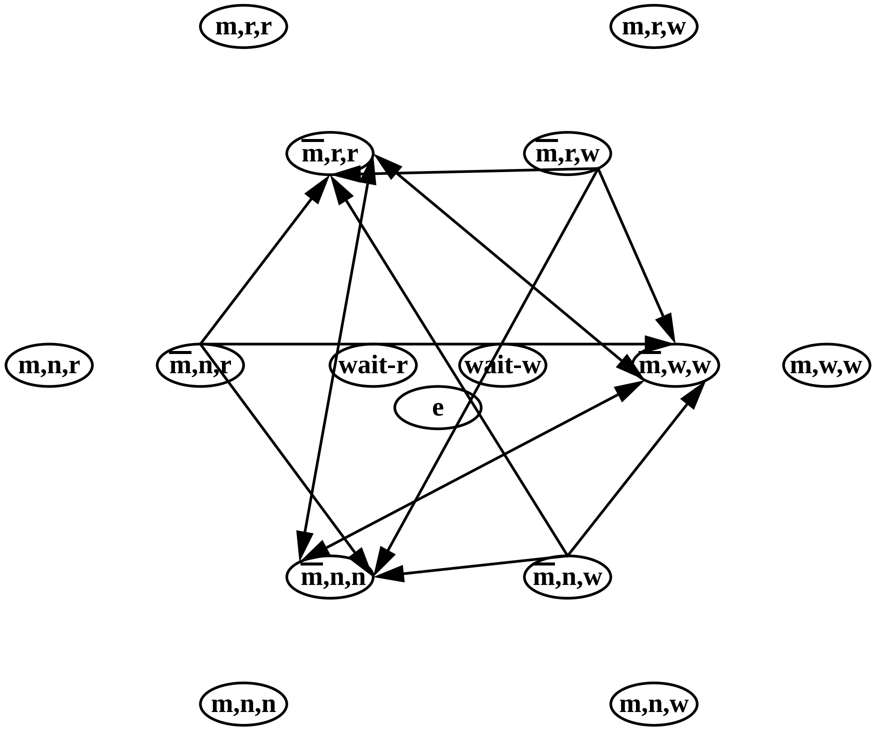

FIGURE 7 **Lock, Clean Transitions—un-modified states**

A *lock* request causes the kernel to forget what accesses it had before (or had requested); the threads currently taking advantage of the old access and those waiting for more access will again attempt to access the page, possibly causing new unlock messages to be generated. As such, some of the transitions here are short-lived; for example, the transi-

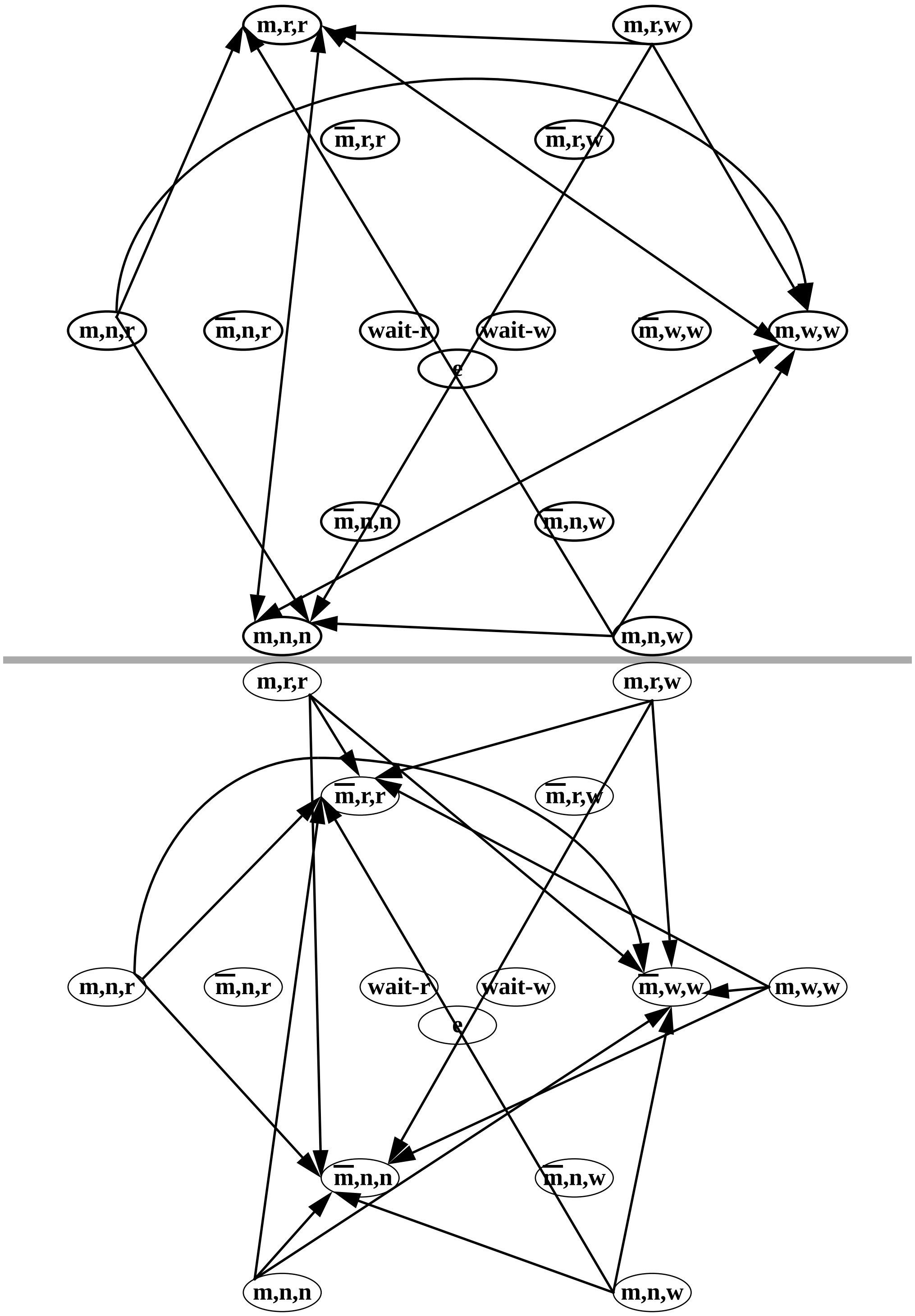

FIGURE 8 Lock, no clean (upper); Clean (lower) Transitions—modified states

tion from <**m,r,w**> to <**m,n,n**> will probably be followed by a thread "poke", changing to state <**m,n,w**>. The lock request itself will receive a **memory_object_lock_completed** message. The *clean* transitions are very much similar to the simple lock requests. The big difference is that (copies of) modified (and possibly precious pages) will be returned to the manager (**memory_object_data_return** or **memory_object_data_write**) prior to the **memory_object_lock_completed** message. Note that the *clean* operation removes the modified flag from the page.

FIGURE 9 shows the various kernel-initiated transitions resulting from "peek" or "poke" operations by clients. The dotted lines show kernel and manager initiated flush (evict) operations.

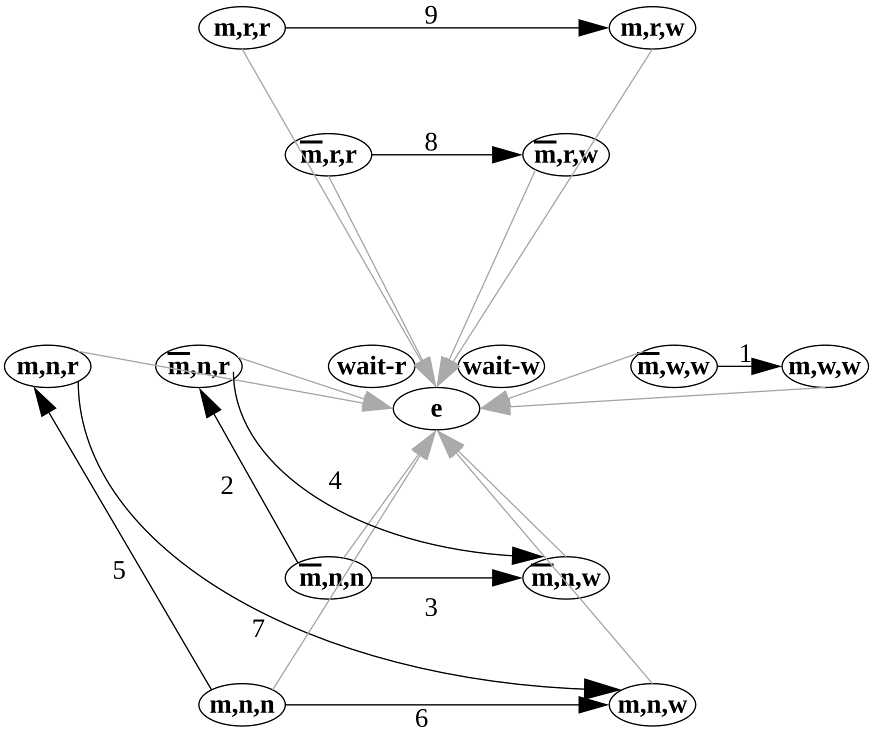

FIGURE 9 **Peek, Poke, Evict and Flush Transitions**

- 1—Some thread "pokes" the page. This is the only transition that sets the *modify* flag (the page must have write access). Note, though, that access may be revoked from the page later, creating a modified page with less than write access.
- 2 and 5—Some thread "peeks" at the page with no access. The kernel sends a **memory_object_data_unlock** message requesting read access.
- 3 and 6—Some thread "pokes" the page with no access. The kernel sends a **memory_object_data_unlock** message requesting write access.
- 4 and 7—Some thread "pokes" the page with no access while the kernel is waiting for read access. The kernel sends a new **memory_object_data_unlock** message requesting write access.

- 8 and 9—Some thread “pokes” the page with only read access. The kernel sends a **memory_object_data_unlock** message requesting write access.

A page may be evicted from any in-memory state (those other than **e**, **wait-r** and **wait-w**) as long as the page isn’t *wired*. The action of evicting the page transitions it to the **e** state; the (possibly modified) page will be returned to the manager in a **memory_object_data_return** (or **memory_object_data_write**) message if it is precious or modified.

The *flush* option to the **memory_object_lock_request** also transitions from all resident states to the **e** state. The kernel will return the page as if it had been evicted if the *clean* option is also specified and if the page is modified or *precious*, followed by a **memory_object_lock_completed** message.

Virtual Copy Optimizations

There are two situations in the Mach system in which a range of memory is *logically* copied.

- When a memory range has the VM_INHERIT_COPY inheritance attribute and a new task is created from this task.
- When a memory range is passed out-of-line in a Mach message. (This includes **vm_read**, **vm_write** and **vm_copy** operations.)

Each of these situations occurs frequently. Copy inherited memory is used to support the POSIX **fork** semantics for new process creation. Out-of-line memory, although uncommon in the normal message case, is very important to supporting external memory managers, file systems in particular, and the device interface to be specific.

It is reasonable for these operations to be defined as logical copies (instead of direct sharing, for instance) because the Mach virtual memory provides virtual copy optimizations. With these optimizations, the memory is not copied outright; memory is copied in a lazily evaluated way, only when data needs to be copied. Data copying can be deferred for a variety of reasons:

- Some of the data is not actually referenced.
- Neither task modifies some of the data, so they can effectively share the same memory image of the un-modified data.
- The task requesting the data copy deletes its mapping, thereby allowing the kernel to consider the copy a move, which can be optimized into page re-mappings (“page stealing”), instead of page copies. A related possibility is that the “receiver” of the data copies it again (and deletes its mappings) without even looking at the data.

On the one hand, these optimizations are purely internal, and therefore not part of the Mach kernel semantics that this document describes. However, as a practical matter, the presence of these optimizations is well known, and an integral part of the utility of many of the interfaces (the device and external memory manager interfaces, especially). Also, the fact that the kernel uses the virtual memory system for out-of-line data allows the various kernel primitives that return lists to do so by returning out-of-line data, which is actually accomplished by internal **vm_allocate** operations. As such, these optimizations form an important part of the specification of the interfaces, namely, the performance characteristics.

Virtual Copy Dialog

When a range of memory is to be copied, the kernel creates a new memory cache object to represent the virtually copied pages. This new memory cache object (the copy destination) may well share actual physical memory pages with the old memory cache object (the copy source), assuming that these pages are not actually modified. The old object is never affected by this virtual copy. It will continue to be associated with the same abstract memory object and the same associated memory manager. Its pages are its own, to use as it sees fit.

Although the new memory object receives virtual copies of all of the pages in the copied range of the old memory object, the visible mechanics of this operation are not so simple, and are controlled by the value of the copy strategy (set by **memory_object_ready** or **memory_object_set_attributes** when the object is initialized, or by **memory_object_change_attributes** or **memory_object_set_attributes** thereafter) for the old memory object, as discussed below. Changing the copy strategy for a memory object can cause a dramatic flurry of activity as the kernel adjusts to the requirements of the new strategy. For this reason, **memory_object_change_attributes** can optionally return a reply (*reply_port* → **memory_object_change_completed**) which can be used for synchronization.

The typical memory object has the property that its pages can only be modified by virtue of being mapped into some task’s address space and being manipulated by direct memory references by that task. If this can only happen on one host, then all modifications to the pages of a memory object will occur on that one host, and be completely visible to the kernel as it maintains its memory cache objects. In this case, the memory manager would set the memory object’s copy strategy to MEMORY_OBJECT_COPY_DELAY, the standard copy-on-write optimization. This causes the new memory object to be built as a temporary object managed by the default memory manager. The behavior of the old and the new objects is as follows:

- Both the new and the old memory objects share pages currently in memory.
- If a page of the new object is referenced that is not memory resident (and not yet “pushed” as below), a message is sent to the old abstract memory object for the data. This request cannot be distinguished by the memory manager from a reference to the old memory object. This fetched page will be shared by the old and the new memory object.
- Whenever a page of the old object is to be modified (by an attempted modify reference to either the old or to the new object (since they are currently sharing physical memory pages)), the kernel first “pushes” the un-modified value of the page into the new object, so that the new object does indeed see a copy of the original data of that page.
- “Pushed” pages are managed by the default memory manager.

The MEMORY_OBJECT_COPY_TEMPORARY strategy also has this effect. This object attribute (temporary) has the additional (unrelated) property that, when terminated as the result of no extant mappings of the object, the kernel discards any cached pages instead of returning them to the memory manager.

If the old memory object can be modified in a way not visible to the kernel on a single host (network shared memory, for example, or direct access by the memory manager), then alternate copy strategies are needed. To see this, consider the following scenario:

- A virtual copy is requested. Some page of the range, however, is not in memory on the copying host.
- The non-resident page of the range, resident on some other host, is modified.
- The new (copy) memory object on the copying host makes a request for the page. It receives the new value, not the value the page would have had at the time of the virtual copy.

To gain control over the semantics in these cases, the kernel provides two alternate copy strategies, MEMORY_OBJECT_COPY_NONE and MEMORY_OBJECT_COPY_-CALL.

MEMORY_OBJECT_COPY_NONE is used when the memory manager is not capable of implementing the correct semantics in an intelligent way. In this case, at the time of the virtual copy, the kernel constructs the new memory object as a default memory manager managed temporary object with its contents explicitly copied (at this time) from the old memory object (thus requesting all pages at this time from the memory manager).

If the memory manager can perform intelligent copy semantics, MEMORY_OBJECT_-COPY_CALL is used. (Important note: This feature is scheduled for replacement. It is un-tested and believed not to work.). This strategy differs from the other two copy strategies in that the new memory cache object, instead of having an associated abstract memory object managed by the default memory manager, has an associated abstract memory object maintained by the same memory manager as that which manages the old memory object. This is done as follows:

- The kernel creates a port to represent the new abstract memory object. The receive right for this port is sent via *old_abstract_memory_object* → **memory_object_copy**., which informs the memory manager of the existence of a new abstract memory object which is to be a virtual copy of the specified old abstract memory object. This call also includes the associated old memory cache control port.
- The kernel performs a *new_abstract_memory_object* → **memory_object_init** call to initialize the new memory object. This informs the memory manager of the name and control ports for the memory cache object associated with the new memory object. The memory manager responds in the usual way.

The memory objects are then maintained as follows:

- Both the new and the old memory objects share pages currently resident in memory.
- Whenever a page of the old object is to be modified (by an attempted modify reference to either the old or to the new object (since they are currently sharing physical memory pages)), the kernel first “pushes” the un-modified value of the page into the new object, so that the new object does indeed see a copy of the original data of that page.

- If a page of the old object is referenced that is not memory resident, a message is sent to the old abstract memory object for the data. This fetched page will appear only in the old memory object.
- If a page of the new object is referenced that is not memory resident (or “pushed”), a message is sent to the new abstract memory object for the data. This fetched page will be “pushed” onto the new memory object, and will not appear in the old memory object. If the memory manager decides that the value of the old memory object’s page can be used, the memory manager may respond with **memory_object_data_unavailable**, which then sends a message to the old abstract memory object, or uses its page if resident.
- “Pushed” pages are not managed by the memory manager; that is, no **memory_object_data_write** or **memory_object_data_return** messages will be sent to the new abstract memory object. (It is not known whether this works correctly or not.)

Default Memory Manager

The default memory manager is, in most regards, simply an external memory manager. It provides backing storage for anonymous memory (**vm_allocate**, copy memory...). A very significant property of the default memory manager is that it is the memory manager of last resort, one which cannot fail. Since no memory manager can provide paging for it, its data writing path is completely wired in memory, and all memory passed to it is wired as well. This memory manager must promptly deal with its memory, and discard it when paged out to backing storage since it is effectively wired.

The default memory manager for a host is set by *host_control* → **vm_set_default_memory_manager**. This is initially the kernel’s internal memory manager.

Memory backed by the default memory manager can be created in a variety of ways. These creations do not involve the default memory manager directly, so the kernel must inform the default memory manager explicitly about new default memory objects. This is done with *abstract_memory_object* → **memory_object_create**. A memory object to be managed by the default pager can be created by the privileged *abstract_memory_object* → **default_pager_object_create.**

The default memory manager provides backing storage for temporary memory objects created by the kernel to represent virtual copy ranges. These temporary objects (except in the case of the MEMORY_OBJECT_COPY_CALL strategy, which does not work correctly) have an additional operation applied to them. When a page from the original object is “pushed” onto them, this pushed data is supplied to the default memory manager with *abstract_memory_object* → **memory_object_data_initialize**. The reason for this additional primitive is that the kernel does not completely track the extent to which it pushes pages onto the copy. (If the copy page is paged-out, a subsequent modification of the original object will push the modified original page again.) As such, it is possible for the default memory manager to receive more than one “push” message (**memory_object_data_initialize**) for a page. The manager must ignore all but the first of these. Note that **memory_object_data_initialize** is called only when a page is pushed onto the copy; if the copy’s pages are themselves modified, the modified pages will be sent to the manager with **memory_object_data_write** or **memory_object_data_return**.

It can happen that the default memory manager will be asked for a page of a copy that has never been “pushed” to it. The manager’s response in this case, as it would be for not-yet-existing pages, is to call **memory_object_data_unavailable**. Instead of creating a zero page, this call, in such a case, will copy the appropriate page from the original object (it follows that the original page has not been modified since no initialize message has been seen).

The default memory manager maintains various (disk) partitions as backing storage. The extent of the default partition (currently the only one) can be found with *abstract_memory_object* → **default_pager_info** (using a default pager object port).

Miscellaneous Operations

A service related to memory objects and memory managers is provided by **device_map**, which allows a portion of a task’s address space to represent a physical device. This is discussed under Devices.

CHAPTER 6 Physical Resource Management

The Mach kernel virtualizes the various resources it maintains. None the less, it is still true that everything depends on the underlying physical resources of the system, and clients are often aware of this, not only to the extent that resource shortages occur, but also to the extent that clients have some direct influence over physical resources.

Physical Memory

Needless to say, the Mach kernel itself uses physical memory. The various kernel objects and associated data structures occupy physical memory. It is a hardware and implementation issue as to which of these structures can be swapped or paged out of memory. Currently, clients have no control over this memory, except to the extent that they create kernel managed entities.

The majority of the physical memory of the system forms a single paging pool. This pool of pages forms a cache for the virtual memory system. The set of pages that reside in physical memory at any given time is decided by the page replacement algorithm, implemented in the kernel. With the exception of the privileged **vm_wire** call, clients have no control over this algorithm. Even external memory managers have no influence; if they do not respond fast enough to a request to write a page, the default memory manager will be used to move the page from physical memory to system paging storage.

When a memory object is no longer referenced, the kernel normally frees all of its physical memory pages. A memory manager can mark an object’s pages as *cacheable*, meaning that the object’s pages will be moved to a free list, but not destroyed. This would tend to be specified for memory objects that persist.

Processors

Each physical processor that is part of a host (and capable of executing threads) is named by a processor port. Although significant in that they perform the real work, processors are not very significant in the Mach scheme of things other than as members of a processor set. It is a processor set that forms the basis for the pool of processors used to schedule a set of threads, and that has scheduling attributes associated with it.

Significant operations on processors are to affect their processor set membership, and to physically control the processor. The list of processor ports for a given host is obtained by *host_control* → **host_processors**.

FIGURE 10 shows the client visible processor and processor set structures. The processor set control port provides access to the contained tasks, threads and processors (only one shown here). The set of all processors is also found via the host control port.

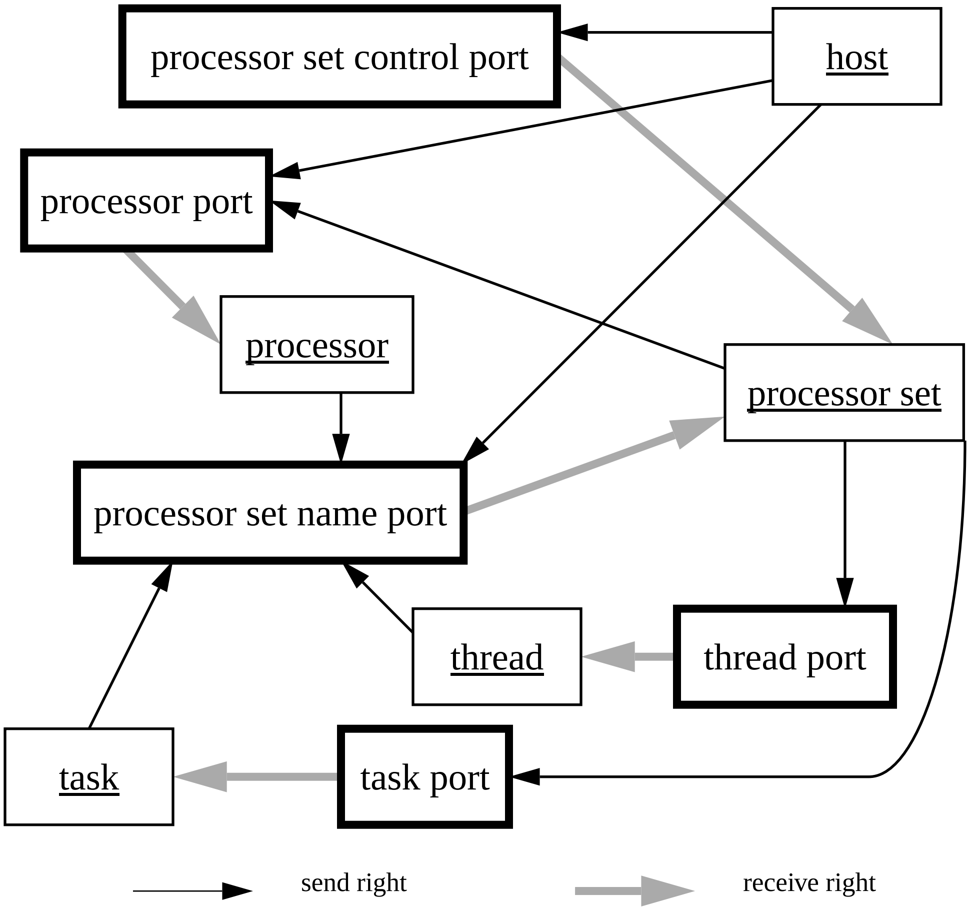

FIGURE 10 **Processor Structures**

The following operations exist:

- *processor* → **processor_assign** — Assign a processor to a processor set
- *processor* → **processor_control** — Perform machine specific control operations
- *processor* → **processor_exit** — Stop a processor
- *processor* → **processor_get_assignment** — Return the name port for the processor set containing the processor
- *processor* → **processor_info** — Return machine properties
- *processor* → **processor_start** — Start a processor

Processor Sets

Processors are grouped into processor sets. A given processor belongs to at most one processor set. A processor set forms a pool of processors used to schedule the threads assigned to that processor set. A thread is assigned to only one processor set. A processor set exists as a basis to uniformly control the schedulability of a set of threads. The notion also provides a way to provide coarse allocation of processors to given activities in the system.

Set Maintenance

A host possesses a set of processor sets. There is always a default processor set, which is automatically created when a host is initialized. The name port for this default processor set can be obtained by *host_name* → **processor_set_default**. The list of name ports for all of the processor sets can be obtained with *host_name* → **host_processor_sets**.

A control port for a processor set can be obtained only via the creation of a processor set, or via the privileged *host_control* → **host_processor_set_priv** call, which translates a processor set name port into its control port.

All other processor sets must be created explicitly, with *host_name* → **processor_set_-create**. This call returns both the name and control ports for the new processor set. A new set is empty. The set is destroyed with *processor_set_control* → **processor_set_destroy**. Any processors currently assigned to this set will be re-assigned to the default processor set.

A processor is added to a processor set with *processor* → **processor_assign**. This removes the processor from its current set. The current assignment of a processor (its processor set name port) is obtained with *processor* → **processor_get_assignment**.

Information (such as scheduling parameters) for the processor set are obtained with *processor_set_name* → **processor_set_info**.

Task and Thread Assignment

By default, a task is *assigned* to the default processor set. Assigning a task to a processor set means that, by default, any new threads in that task will be assigned to that processor set. Being assigned to a processor set means that the thread will only execute on processors within that set.

A task is assigned to a processor set with *task* → **task_assign**, a thread with *thread* → **thread_assign**. These calls require a control port to the processor set, which most tasks do not hold. For simple assignment to the default processor set, there is *task* → **task_assign_default** and *thread* → **thread_assign_default**, which need no such port. The current assignment can be found with *task* → **task_get_assignment** and *thread* → **thread_get_assignment**, both of which return a processor set name port.

The current set of tasks / threads assigned to a processor set is found with *processor_set_-control* → **processor_set_tasks** / *processor_set_control* → **processor_set_threads**.

Kernel Threads and Tasks

For the sake of its own operation, the kernel creates *kernel threads* that execute purely within kernel context to provide various support functions. For example, page eviction (the "page-out daemon"), asynchronous I/O post-processing, thread reclamation and scheduler priority computations are performed by dedicated threads rather than being executed in interrupt (or software interrupt) context. Users of the system (including privileged ones) have no direct control over these internal threads. For the sake of giving these threads a task context, the kernel constructs a *kernel task* to contain them. This kernel task has no user address space.

The default pager forms its own kernel task with its own set of kernel threads to perform the processing. (The page-out thread is part of the main kernel task, not the default pager task.)

These two kernel internal tasks are under kernel control. However, since the contained threads must be scheduled as are any other threads, they are linked into the processor set lists, initially assigned to the default processor set. Thus, the result from **processor_set_-tasks** will list these two tasks as the first tasks in the list (by convention) and **processor_set_threads** will list the kernel internal threads (in no determinate order).

Since these kernel entities have no user context, most operations against their task or thread ports will fail. They can be assigned different scheduling priorities and assigned to different processor sets. Attempting to terminate or suspend these threads is undefined (but likely to cause system failure).

Scheduling Control

Processor sets achieve two goals: they provide for coarse allocation of processors to a set of tasks (an application); they also provide the handle for dictating scheduling policies.

Any given thread has associated with it a scheduling policy to use, and various parameters that influence that policy, the priority being the most obvious.

Any given processor set supports the scheduling of its threads according to the scheduling policies allowed for that processor set. The set of policies enabled for a processor set are controlled by *processor_set_control* → **processor_set_policy_disable** and *processor_set_control* → **processor_set_policy_enable**. The currently enabled set can be found with *processor_set_control* → **processor_set_info**.

Tasks do not have a scheduling policy associated with them. Threads have a scheduling policy set via *thread* → **thread_policy**. When a thread is assigned to a processor set, it maintains its policy unless the processor set does not have it enabled, in which case the thread's policy is set to time-sharing. However, disabling a policy for a processor set does not affect any threads currently in that set (unless an option to the **processor_set_-policy_disable** call is used).

Threads have three priorities associated with them by the system:

- A priority value which can be set by the thread to any value up to a maximum priority. Newly created threads obtain their priority from their task. This is set by *task* → **task_priority.** This value can be affected by *thread* → **thread_priority**.
- A maximum priority value which can be raised only via privileged operation so that users may not unfairly compete with other users in their processor set. Newly created threads obtain their maximum priority from that of their assigned processor set. The maximum priority for a processor set is set by *processor_set_control* → **processor_set_max_priority**. A "privileged" thread may raise its maximum priority if it holds the processor set control port with *thread* → **thread_max_priority**.
- A scheduled priority value which is used to make scheduling decisions for the thread. This value is determined on the basis of the user priority value by the scheduling policy (for timesharing, this means adding an increment derived from CPU usage). This value can be affected by *thread* → **thread_priority**.

Nodes

The individual uniprocessor or multiprocessor *nodes* in a Mach multicomputer system are generally maintained in a transparent fashion. As such, there are very few operations defined for specific nodes.

Each node is allowed a small set of node specific ports. A few are set aside for client use (such as a node specific name server port). The kernel implements three node specific ports:

- A "host" control port, useful for deriving processor set and other ports which are associated with specific nodes but which can only be obtained from operations requiring a host control port.
- A corresponding "host" name port.
- A device master port, used to obtain node specific devices.

A node special port is obtained with *host_control* [*node*] → **norma_get_special_port** and set with *host_control* [*node*]→ **norma_set_special_port**.

The only other node visibility is the location of tasks. Currently, a task is created on a node and will not migrate from that node. By default, a task is created on the same node as the task used as the parent in the **task_create** call. This default can be changed with *task* → **task_set_child_node**; this call specifies the target node for future children. A task may also be created explicitly on a given node with *task* → **norma_task_create**.

task (*port*) → **norma_port_location_hint** provides a guess of a port's current location. For port's whose receive right never moves this will return the correct node, otherwise it returns either the correct node or a node at which the port once was.

Hosts

Each multiprocessor (or sets of multiprocessors in a multicomputer) within a networked Mach system runs its own instantiation of the Mach kernel. The *host* machine is not generally manipulated by client tasks. But, since each host does carry its own Mach kernel, each with its own port space, physical memory and other resources, the executing host is visible and sometimes manipulated directly. Also, each host generates its own statistics.

The most likely way in which a task may be aware of the presence of multiple Mach kernel instantiations is in external memory managers. Each kernel sends its own messages to a given external memory manager that is trying to manage memory mapped on more than one host.

Each host has its own physical memory pool, set of devices, default memory manager, set of processor sets (and, therefore, assigned threads and tasks) and time.

The name port to the host on which a thread is executing (on which its containing task was created) is returned by the **mach_host_self** trap.

Each host also has a control port used for its manipulation. For historic reasons, the control port for the host is called its privileged host port. No primitive returns this port. This port is supplied to the bootstrap task as described below.

FIGURE 11 shows the client visible host structures. Access to the host control port allows access to the processor set and processor ports.

The following host specific calls are provided:

- *host_control* → **vm_set_default_memory_manager** — Set the default memory manager.
- *task* → **vm_statistics** — Return host-wide memory usage statistics. (Note that these statistics are returned given any task port on the host.)
- *host_control* → **host_adjust_time** — Make a clock adjustment.
- *host_control* → **host_get_boot_info** — Return operator supplied boot-time information.
- *host_name* → **host_get_time** — Return the current time.
- *host_name* → **host_info** — Return host information.
- *host_name* → **host_kernel_version** — Return the kernel version.
- *host_control* → **host_reboot** — Re-boot, with options.
- *host_control* → **host_set_time** — Set the current time.
- *host_name* → **host_processor_sets** — Return name ports to the defined processor sets.

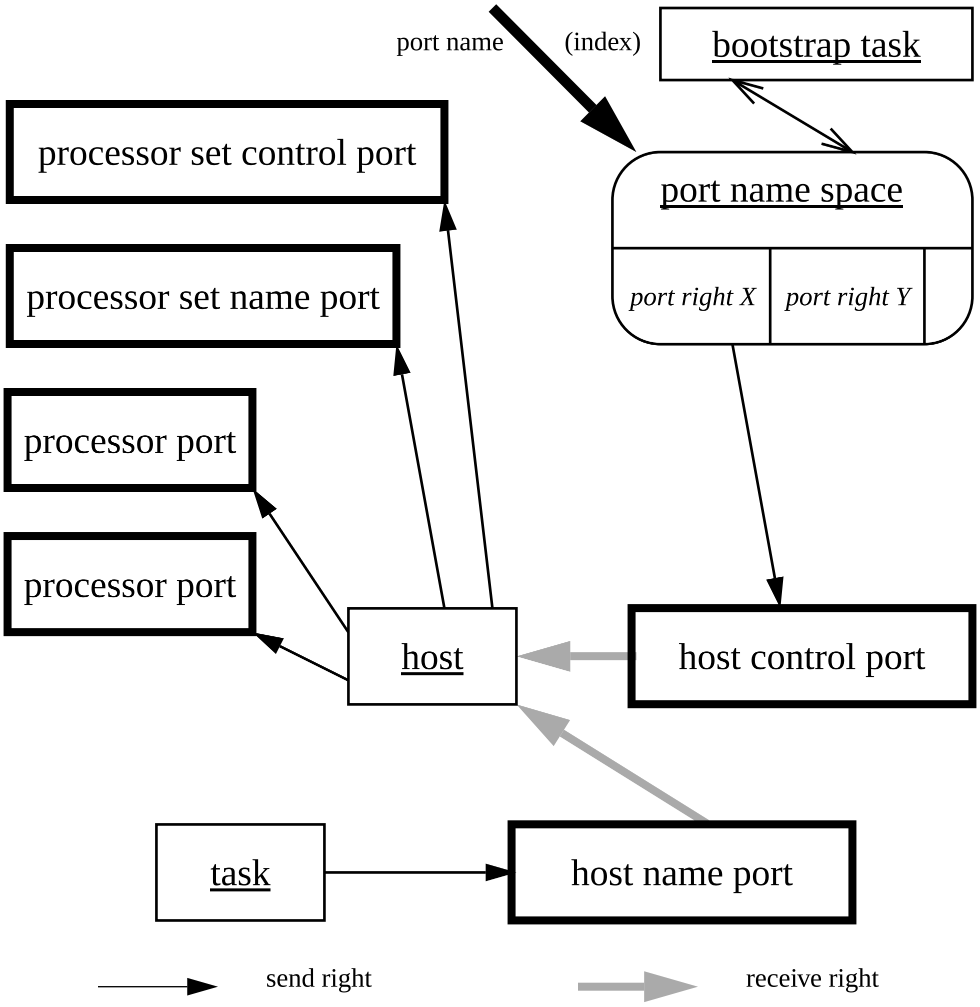

FIGURE 11 **Host Structures**

- *host_control* → **host_processors** — Return control ports for the host's processors.
- *host_control* → **host_processor_set_priv** — Convert a processor set name port to a control port.

Devices

The Mach kernel exports a very simple interface to its devices. When initialized, the Mach kernel builds an internal table that lists each device. It exports a single port, the *device master* port, which is responsible for allocating devices. A task that holds send rights to the device master port may request the kernel to *open* a device, returning a port that provides access to that device. Operations on that port then manipulate the device, until it is *closed*.

In as much as that Mach messaging semantics are asynchronous, the device interface is asynchronous. A client sends a message to a device; a reply message indicates status or data returned. There exist, though, MIG generated stubs that use a message send/receive pair to provide a synchronous interface. In keeping with the style of this document (which is to list the synchronous RPC interfaces when they exist), the synchronous versions are shown here.

Given access to the device master port, the port controlling a named device is returned by *device_master* → **device_open**. (Normally, only some device mastering task would hold the device master port.) The device is freed from use by *device* → **device_close**.

There is no primitive that returns the device master port. This port is provided to the bootstrap task as described below.

FIGURE 12 shows the client visible device structures. Access to the device master port provides access to all devices.

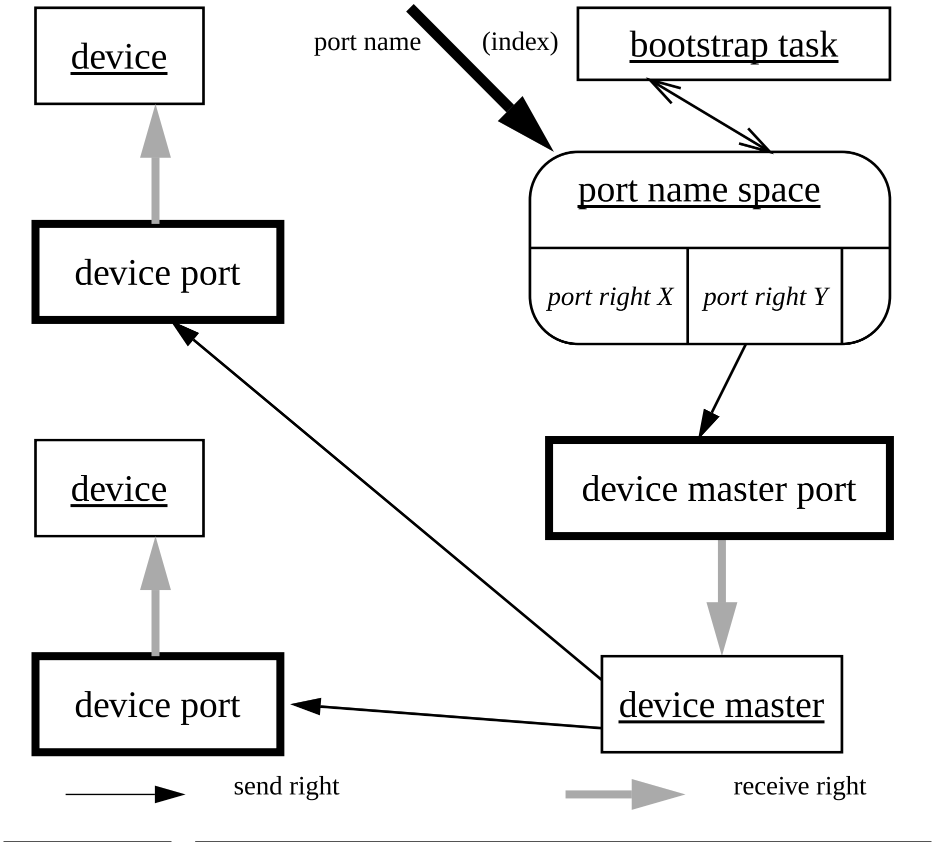

FIGURE 12 **Device Structures**

Each device is modeled as a set of records of some indeterminate size. Each device defines these notions for itself. There is a read and a write interface, both in two forms: one which uses out-of-line virtual memory managed space, and one that transmits data “in-line” (inband) in the message (efficient for small transfers). These calls are *device* → **device_read**, *device* → **device_read_inband**, *device* → **device_write** and *device* → **device_write_inband**.

Each device also defines some status information, read by *device* → **device_get_status** and set by *device* → **device_set_status**.

There are two very device specific functions defined by the kernel for special purpose devices.

- *device* → **device_set_filter** — This call establishes an input filter for all data that appears upon the device. The filter is expressed as a small “program” for a simple stack machine implemented by the driver. This is used to filter incoming network packets, sending asynchronous messages carrying the packets that pass the filter to the protocol server task specifying this filter for this device. Multiple packet filters can be specified (possibly by multiple tasks) and each packet may well be sent to multiple recipients.
- *device* → **device_map** — This call is similar to **vm_map**. It introduces a new memory range in the task’s virtual address space. The external memory manager for this range is the device driver, which can provide any illusion of the device. The typical use of this call is to map a workstation’s frame buffer into the display server’s memory.

A “mapped” device (one associated with a shared memory window established by **device_map**) provides support for user space device drivers. The shared memory window can provide some limited access to device hardware registers to allow for direct user space manipulation of the device or perhaps direct interaction with the kernel device driver so that each I/O operation does not require a Mach IPC message (and the associated costs, especially the copy of data). Mapped devices provide an interrupt service via the **evc_wait** system trap and device driver defined event counts.

The event count service exists for two reasons:

- A device driver interrupt routine cannot call Mach IPC to send a message indicating the interrupt.
- The use of event counts is considerably cheaper than IPC messages.

The event count service defines one or more event objects, named by task local event IDs. Each of these event objects has an associated event count, initially zero. Whenever the associated event occurs (typically a device interrupt), the event count is incremented. If this count is zero when **evc_wait** is called, the calling thread waits for the next event to occur. Only one thread may be waiting for the event to occur. If the count is non-zero when **evc_wait** is called, the count is simply decremented without causing the thread to wait. The event count guarantees that no events are lost.

When a device interrupt occurs, the kernel device driver would place device status in the shared memory window and signal the associated event. The user space device driver would normally be waiting with **evc_wait**. The user thread wakes, processes the device status, typically interacting with the device via its shared memory window, then waits for the next interrupt.

Access to Privileged Ports

The holder of the host control port can obtain send rights to any port on the system (with the exception of device ports), including “privileged” ones, such as the processor set control ports. The basic privilege mechanism provided by the kernel is the restriction of privileged operations to tasks holding control ports, with the various control ports being derivable only from the host control port.

Likewise, the holder of the device master port can obtain any device port, and these ports can only be derived from the master device port.

The integrity of the system depends on the close holding of these two ports. There is no kernel operation that returns these ports. Only the bootstrap task (the first task) is supplied with these ports (although that task may subsequently give them out) in the manner described below.

When the bootstrap task is created, its bootstrap port is set to be a special kernel port (one whose receiver is the kernel). This bootstrap port will respond to one and only one request for service. One of the first things that the bootstrap task is to do when it starts is to send a message to this port (the message ID doesn’t matter but 999999 is the convention). The reply from this request will contain simply two port rights, the host control port and the device master port (in that order).

(Implementation detail: It is important for the bootstrap task to make this request, not only so that it will function, but also because the kernel thread waiting for the request is the one that becomes the initial default pager thread.)