OSF Mach *Final Draft* Kernel Principles

Open Software Foundation and
Carnegie Mellon University

Keith Loepere

May 3, 1993

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

Books in the OSF Mach series:

OSF Mach Kernel Principles

OSF Mach Kernel Interfaces

OSF Mach Server Writer's Guide

OSF Mach Server Library Interfaces

Revision History:

Revision 3	Draft: October 7, 1992
Revision 3.1	Draft: November 13, 1992
Revision 3.2	Final Draft: May 3, 1993

Copyright 1991-1993 by the Open Software Foundation, Inc. and Carnegie Mellon University.

All rights reserved.

Permission to reproduce this document without fee is hereby granted, provided that the copyright notice and this permission notice appear in all copies, derivative works or modified versions.

OSF MAKES NO WARRRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Contents

CHAPTER 1 **Introduction** 1
Kernel Abstractions 2
Structure of this Document 5

CHAPTER 2 **Architectural Model** 7
Elements 7
Threads 10
Tasks 10
Security Port 11
Ledgers 12
Ports 12
Messages 12
Message Queues 13
Port Rights 13
Port Name Space 13
Port Sets 14
Virtual Address Spaces 14
Abstract Memory Objects 15
Memory Object Representatives 16
Memory Cache Objects 16
Processors 16
Processor Sets 17
Hosts 17
Nodes 17
Clocks 18
Kernel Devices 18

CHAPTER 3 **Threads and Tasks** 19
Threads 19
Tasks 25

CHAPTER 4 **Ports, Rights and Messages** 29
Ports 29
Messages 31
Message Queues 32
Port Rights 32
Port Name Space 35

Port Sets 37
Message Transmission 37

CHAPTER 5 **Virtual Memory Management** 43
Virtual Address Spaces 43
Memory Objects. 46

CHAPTER 6 **Physical Resource Management** 65
Physical Memory 65
Ledgers. 66
Processors 67
Processor Sets 68
Clocks 71
Hosts 73
Nodes 74
Kernel Devices 75

CHAPTER 7 **System Initialization** 79
Bootstrap Task Properties 79
Access to Privileged Ports 80

APPENDIX A **Function Index** 81

Tables and Figures

FIGURE 1 **Thread Structures** 20
FIGURE 2 **Task Structures** 25
FIGURE 3 **Port Structures** 30
FIGURE 4 **Port Right Structures** 33
FIGURE 5 **Virtual Memory Structures** 48
FIGURE 6 **Page-in Transitions** 55
FIGURE 7 **Lock, with or without Clean (un-modified) Page Transitions** 57
FIGURE 8 **Lock, without Clean (modified) Page Transitions** . . . 58
FIGURE 9 **Lock, with Clean (modified) Page Transitions** 59
FIGURE 10 **Peek, Poke, Evict, Flush Page Transitions** 60
FIGURE 11 **Processor Structures** 68
FIGURE 12 **Host (and Clock) Structures** 74
FIGURE 13 **Kernel Device Structures** 76

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 developed as a new foundation for operating systems. It provides extensible 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.

In versions 2.5 and earlier, Mach was combined with Unix[1] to deliver a complete operating environment. Version 3, however, provides Mach as a pure kernel with no other operating system functionality in kernel space. Particular operating environments are

provided by means of user space servers. To date, servers have been prototyped for BSD, SVR4, Sprite, OSF/1 and DOS.

Mach as a foundation technology for diverse operating environments 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, clocks).

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 — Collection of data objects.
- Memory object — Internal unit of memory management.

1. Unix is a registered trademark of Unix System Laboratories.

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 (POSIX 1003.4a, 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 mul-

tiple 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 the elements of this caching itself; it endeavors to allow user mode tasks the ability to participate in these mechanisms.

Unlike 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 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 (bounded length) queue that holds *messages*. A message is a collection of data, memory regions and port rights. A port *right* is a name by which a task that holds the right names the port. 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 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 (a 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.
- *security — Not really a kernel object. The privileged security port is used when setting (changing) task identity.
- *ledger — A resource accounting device that counts and limits the amount of a type of kernel resource that may be consumed.
- *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 collection of data, memory regions and port rights 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 memory managers external to the kernel.
- 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. The abstract memory object port is the port through which the kernel requests action of the memory manager.
- memory object representative—The abstract representation of a memory object provided by the memory manager to clients of the memory object. The representative names the associated abstract memory object and limits the potential access modes permitted to the client.
- *memory cache object — A kernel object that contains the resident state of the memory ranges backed by an abstract memory object. It is through this object that the memory manager manipulates the clients' visible memory image.
- *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.
- *host — The multiprocessor as a whole.
- *node — An individual multiprocessor within a multicomputer.
- *clock — A representation of the passage of time (a time value incremented at a constant frequency).
- *kernel device — Physical device accessible by user mode tasks.

(An asterisk denotes those elements that are kernel implemented resources that can be directly manipulated by threads.)

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. 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 communication 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 information.

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. Some operations require two objects (such as binding a thread to a processor set); these operations show the objects separated by commas. 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. 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. The parenthesized notion,

object (*port*) → **function**

is here used to show that a (privileged) *port* (such as the host control port) is required as a parameter in the message to indicate sufficient privilege to manipulate the object in the particular way.

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 ports 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_overwrite_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 (such as exception 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 security token to be sent with messages from the task.
- A (small) set of task specific ports, identifying the task's kernel port, the task's ledger, default ports to use for exception handling for contained threads and bootstrap ports to name other services.
- System call emulation routine addresses for routines that can be set to 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.
- Setting the security token.
- Suspension and resumption.
- Special port manipulation.
- Manipulation of contained threads.
- Statistical PC sampling of the contained threads.
- Manipulation of the scheduling parameters.

Security Port

All tasks are tagged with a security token, an opaque (from the kernel's point of view) identifier that encodes in some way the identity and other security attributes of the task. This security token is included as an implicit value in all messages sent by the task. Servers can use this sent token as an indication of the sender's identity for use in making access mediation decisions.

A task inherits the security token of its parent. Since this token is to be used as an unforgeable indication of identity, privilege is required to change this token. This privilege is indicated by presenting the host *security* port.

The reserved value of KERNEL_SECURITY_TOKEN indicates the kernel's identity. All messages from the kernel carry the kernel identity.

Ledgers

A ledger is a resource accounting device that counts and limits the various kernel resources consumed by an object. All first-class (port named) dynamically created kernel objects (with the exception of threads since they are bound to a task) are associated with a ledger which represents a pool of kernel resources from which that object's resources are drawn.

The operations meaningful for ledgers are:

- Creation of a sub-ordinate ledger. The new ledger draws its resources from the parent.
- Use of the ledger as the source of resources for the creation of a kernel object.
- Transfer of resource (limits) between a parent and child ledger.
- Destruction. This act also destroys all kernel objects drawn from the ledger.

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 state associated with a port is:

- Its associated message queue.
- A count of references (rights) to it.
- Settable limits on the amount of virtual copy memory, port rights and data that can be sent in a message through the port.

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 collection of data, memory regions and port rights 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 of:

- Pure data
- Copies of memory ranges
- Port rights
- Kernel implicit attributes, such as the sender's security token

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. Also, 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.

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. As port rights are manipulated (by referring to their port names) the port names are sometimes themselves manipulated.

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.
- Force flush a range to backing storage.
- Reserving a range (preventing random allocation within the range).

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.
- Synchronization with force flush operations.
- Requests for permission to access pages.
- Termination.

Memory Object Representatives

The abstract memory object port is used by the kernel to request access to the backing storage for a memory object. Because of the protected nature of this dialog, memory managers do not typically give access to the abstract memory object port to clients. Instead, clients are given access to *memory object representatives*.

A memory object representative is the client's representation of a memory object. There is only one operation permitted against such a port and that is to *map* the associated memory object into a task's address space. Making such a request initiates a protocol between the mapping kernel and the memory manager to initialize the underlying abstract memory object. It is through this special protocol that the kernel is informed of the abstract memory object represented by the representative, as well as the set of access modes permitted by the representative.

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 operational attributes.
- 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.
- Force delayed copies of the object to be really copied.
- Indicate that pages sent to the memory manager have been disposed.
- 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.

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:

- Obtaining clock ports.
- Statistics gathering.
- Re-booting.
- Setting the default memory manager.
- Obtaining lists of processors and processor sets.

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 host in such a multicomputer is called a *node* and referenced by a node ID (a number).

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.

Clocks

A clock provides a representation of the passage of time by incrementing a time value counter at a constant frequency. Each host (or each node in a multicomputer) implements its own set of clocks based upon the various clocks and timers supported by the hardware as well as abstract clocks built upon these timers. The set of clocks implemented by a given system is set at configuration time.

Each clock is named by both a name and a control (privileged) port. The control port allows the time and resolution of the clock to be set. Given the name port, a task can:

- Determine the time and resolution of the clock.
- Generate a memory object that maps the time value.
- Sleep (delay) until a given time.
- Request a notification (alarm) at a given time.

Kernel Devices

The Mach kernel exports a very simple interface to its kernel device drivers. 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.

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 processor set or the containing task.

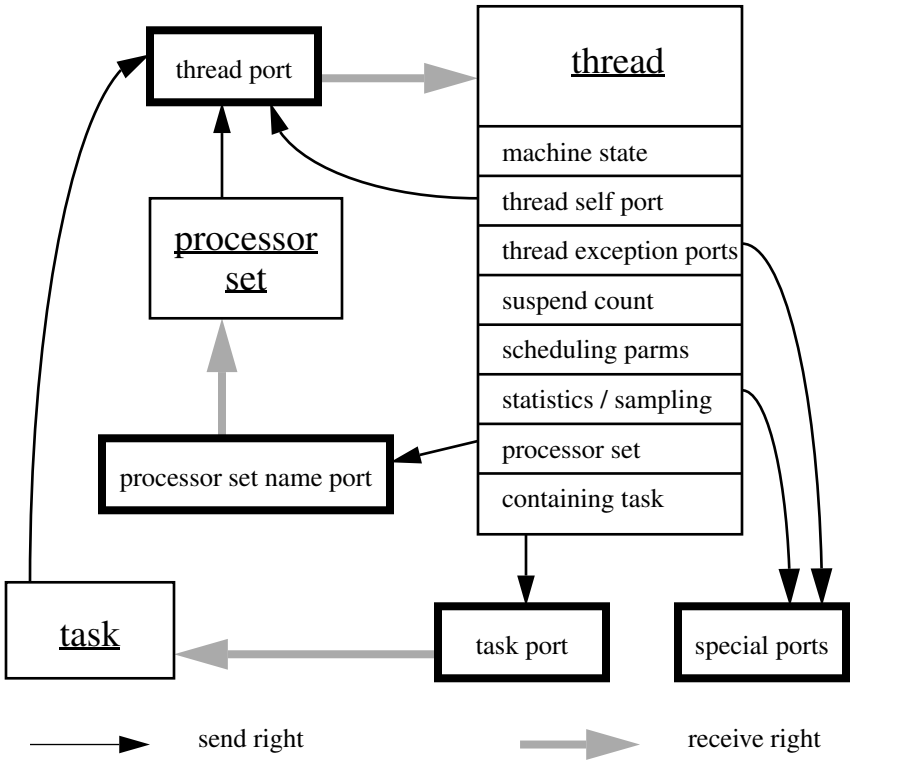

FIGURE 1 **Thread Structures**

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.

Note that the following list of Mach traps perform the specified action only if the exception port for the EXC_MACH_SYSCALL exception class is the host name port.

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 attributes associated with them which are 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 attributes of the thread to be adjusted so that other threads will run, thereby (it is hoped) completing work that blocks this depressed thread. This scheduling 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.

Finally, the **clock_sleep** trap causes the thread to be delayed until a specified time. This delay can be aborted by **thread_abort** causing the **clock_sleep** to generate an error return.

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 known for the task.

Bootstrap Reply Port Trap

The **mach_reply_port** trap is also needed 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. If the task's self port is null (thereby deactivating use of Mach services against the task), this call returns null as well.

Message Send and Receive Trap

The final, and most important trap, is **mach_msg_overwrite_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

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 an exception port. A successful reply to this message causes the thread to continue (in a state possibly altered by **thread_set_state**).

For any given exception, there are two exception ports that may apply:

- A thread specific port for the specific type of exception.
- A task port for the specific type of exception.

The thread specific ports are set with *thread* → **thread_set_exception_ports** and read with *thread* → **thread_get_exception_ports**. The task ports are set with *task* → **task_set_exception_ports** and read with *task* → **task_get_exception_ports**. The *thread* → **thread_swap_exception_ports** and *task* → **task_swap_exception_ports** calls set and return the previous exception ports, performing an atomic swap.

The kernel selects the first of these ports (in the order listed) as the destination of the exception message (assuming it is defined). Whereas a successful reply causes the thread to continue, an unsuccessful reply causes the kernel to send an exception message to the second port. If neither exception message receives a successful reply, the thread is terminated.

The kernel can send various exception message formats, as selected when the exception port was set:

- *exception_port* → **catch_exception_raise**—A message indicating the identity of the faulting thread (the task and thread self ports), the type of exception and status codes.
- *exception_port* → **catch_exception_raise_state**—A message indicating the type of exception, status codes and a flavor of thread state (some register values) selected when the exception port was set. The thread state is both input and output, allowing the reply to change the thread state.
- *exception_port* → **catch_exception_raise_state_identity**—The **catch_exception_raise_state** message with the inclusion of the task and thread self ports.

As specified above, it is possible for a thread to access its exception ports and to be able to receive and generate exception messages which can be useful for programs such as debuggers and other intermediaries. This ability to access the exception mechanism can cause problems for exception servers, though, that need to carefully control exception versus normal processing or processing signals or the like. A server can optionally request, via the exception behavior parameter when setting exception ports, that the exception mechanism for that class be “protected”. If protected, attempts to get the exception port fail, and attempts to abort a thread within exception processing fail.

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

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 always apply to the system call instruction(s).

First of all, several of the possible system call numbers are subsumed for Mach kernel calls. If the exception port for the EXC_MACH_SYSCALL exception class is the host name port, these system calls are executed as described above. If the exception port for this class is any other value, an attempt to execute these system call instructions raises an exception (of class EXC_MACH_SYSCALL) as described above.

The remaining system call numbers are initially undefined. An attempt to execute them results in an exception (of class EXC_SYSCALL) as described above.

At the granularity of a task, an indirection can be specified separately for each system call number. This is done via *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_vector** 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.

As an optimization, the sequence of steps necessary to create a running thread, **thread_create**, **thread_set_state** and **thread_resume**, are combined in the *task* → **thread_create_running** call.

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_abort** aborts non-recoverable operations such as multi-page memory management operations and exception processing. *thread* → **thread_abort_safely**, on the other hand, will refuse to abort non-restartable operations. When **thread_abort** returns, the target thread is suspended at the point of return from the kernel (if it had been in kernel mode) or at the same user instruction where it had been otherwise.
- *thread* → **thread_set_state** can then be safely used.
- *thread* → **thread_resume** restarts the thread.

A thread currently has only one "special" port associated with it. This 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). This port is 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 physical resource management.

- *thread*, *processor_set_control* → **thread_assign**
- *thread* → **thread_assign_default**
- *thread* → **thread_get_assignment**
- *thread* (*processor_set_control*) → **thread_set_policy**
- *thread* → **thread_policy**
- *thread* (*host_control*)→ **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.

thread → **thread_sample** provides support for the Posix **profil** system call. 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 the task port, the task port can be derived from the task's task self port, the contained threads or the containing processor set.

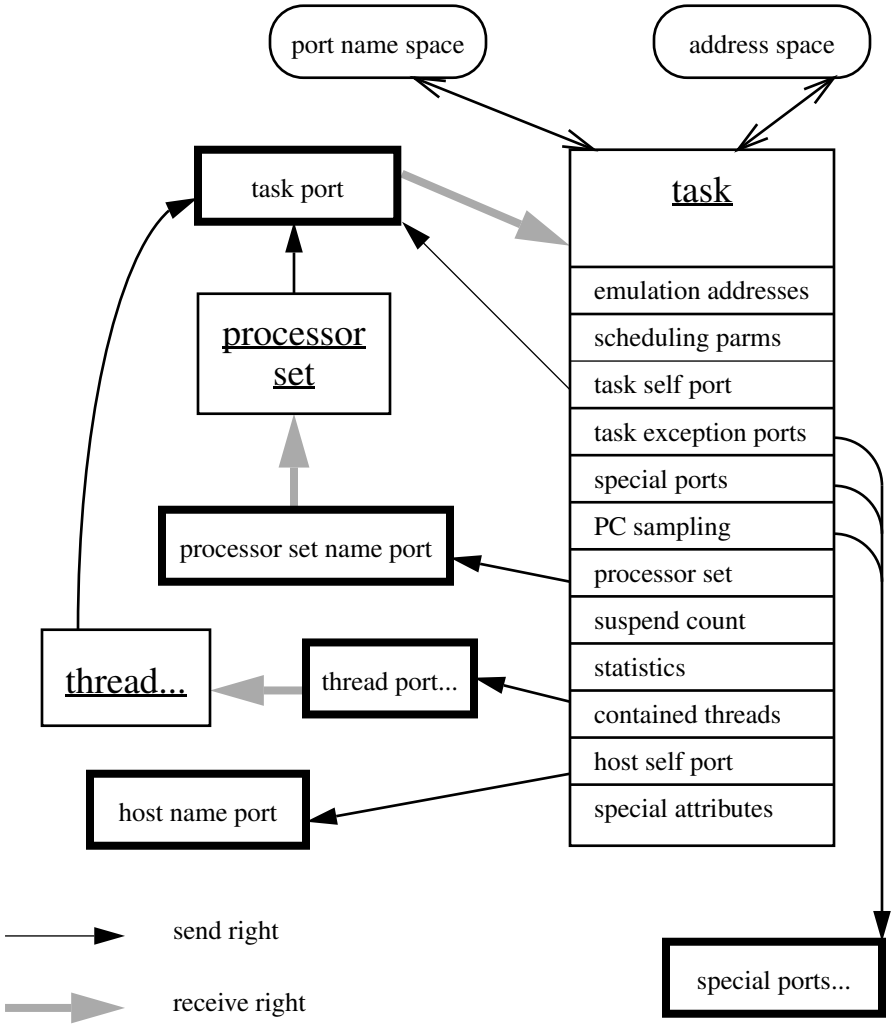

FIGURE 2 **Task Structures**

Life and Death

A new task is created with *task* (*ledgers*) → **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 (parent) 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 parent task. The new task's port name space is empty. The new task inherits the parent task's PC sampling state, special and exception ports.

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 the following "special" ports associated with it:

- A port the task uses 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).
- 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.
- A port used to request information of the containing host, normally the same the host name port.
- A port naming the resource ledger from which the task draws resources. This is normally the same as the actual resource ledger, but can be different if so set.

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 and ledger port).

A task 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.

A task also has exception ports (as described under exception processing) and a program counter profiling port (described below) defined for it, all inherited from the parent task.

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_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*, *processor_set_control* → **task_assign**
- *task* → **task_assign_default**
- *task* → **task_get_assignment**
- *task* → **task_policy**
- *task* (*processor_set_control*) → **task_set_policy**

task → **task_sample** provides support for the Posix **profil** system call. 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.

Identity

Each task is labeled with a security token, not interpreted by the kernel. It is sent in messages sent by the task for use by trusted servers in access mediation decisions concerning the requestor of service. This security token is inherited from the parent task by **task_create**, or explicitly set via *task* (*security*) → **task_create_security_token**. It can be changed only with the privileged *task* (*security*) → **task_set_security_token** call. The security port is a privileged port provided to the bootstrap task whose sole purpose is to support the setting (changing) of task identity.

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 state associated with a port is:

- The associated message queue
- A count of references (rights) to it.
- Port right, out-of-line memory and data size receive limits.

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.

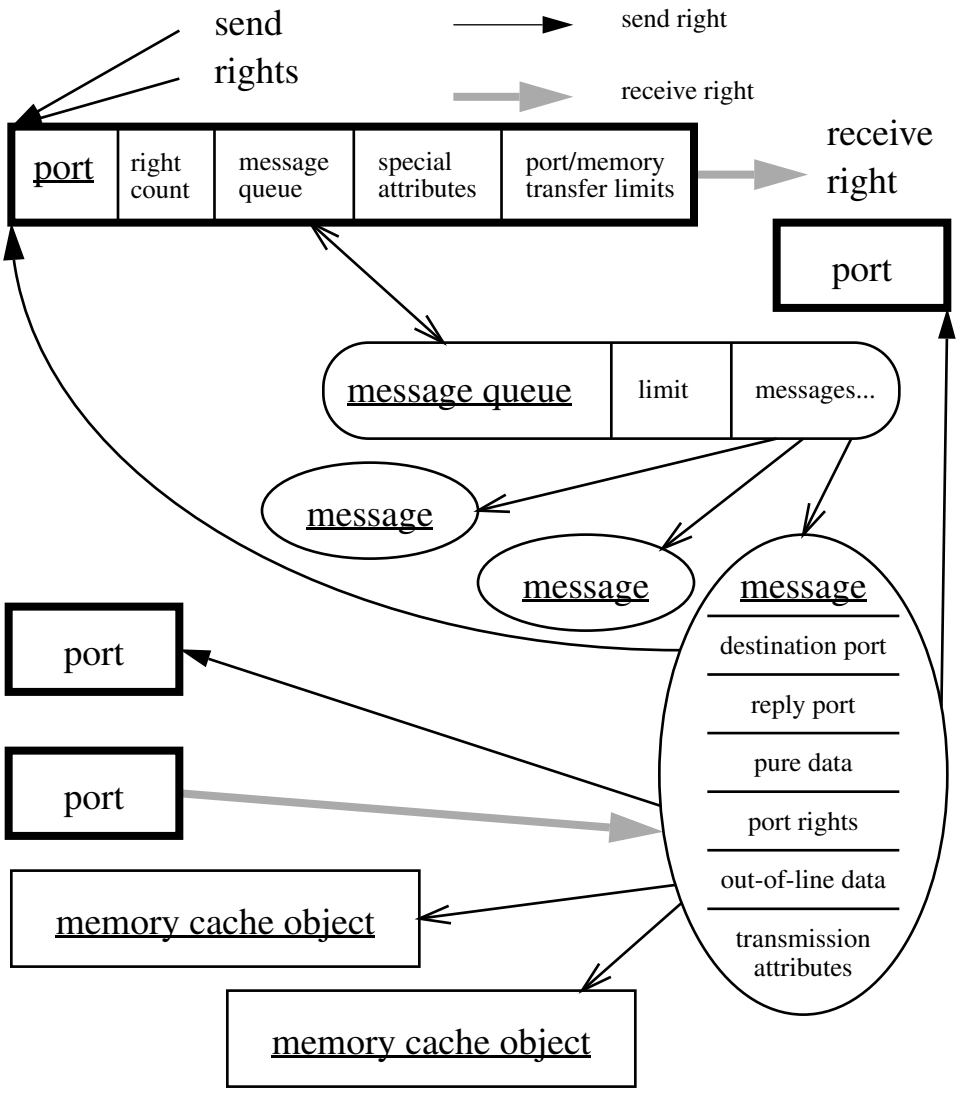

FIGURE 3 **Port Structures**

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

Ports are created implicitly when any other system entity 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.

Most of the attributes of a port are assigned at the time of creation. Some of these attributes (such as the limit on the number of port rights or amount of out-of-line memory that can be received in a message) can be changed with *port_name_space* [*port_name*] → **mach_port_set_attributes**. These attributes can be obtained with *port_name_space* [*port_name*] → **mach_port_get_attributes**.

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 an option to **mach_msg** (MACH_RCV_NOTIFY), as well as with *port_name_space* [*port_name*] → **mach_port_request_notification**.

The resultant dead name notification indicates that a task's port name has gone dead because of the destruction of the named port. The message indicates the task's name for the now dead port. (This is discussed under port name spaces.).

Messages

A message is a collection of data, (out-of-line) memory regions and port rights 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**) followed by a message body. A message also carries attributes implicitly added by the kernel, such as the sender's security token. All messages generated by the kernel itself carry the kernel's identity.

The header specifies a port name for the port to which the message is sent and an auxiliary port name for the port to which a reply is to be sent (if a reply is requested).

The body consists of descriptors of kernel manipulated data (out-of-line memory regions, port rights and port right arrays) and some additional message body data. The message body data normally contains type conversion information as well as random data elements.

If the message is to contain additional port rights or out-of-line memory regions, the MACH_MSGH_BITS_COMPLEX field is set in the header. In this case, the header is followed by the **mach_msg_body** structure, which gives a descriptor count and an array of descriptors naming these elements as well as describing the kernel's processing for these elements.

A descriptor for a single port right names the right as well as any special transformations performed by **mach_msg**, such as moving the right versus making a copy or generating a right from a right of a different type. A descriptor for an "out-of-line" port array also specifies the IPC processing for the set of rights (which must all be of the same type) but specifies the address and size of an array of port names.

An out-of-line data descriptor specifies the size and address of the out-of-line region. The region need not start at the beginning of a page nor contain an integral number of pages; only the specified data is logically transmitted. The descriptor can specify that the

act of queueing the message will de-allocate the memory range (as if by **vm_deallocate**) from the sending task. The sender is given the choice of sending a “physical” or “virtual” copy of the data (discussed under message transmission).

Neither the values in the out-of-line data regions or in the message data are typed. Only the receiver of a message, with the knowledge of the port over which it came and the message data, can interpret (and possibly transform) these data areas.

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_attributes** given the receive right for the associated port. If a message queue is full, no more messages can be queued (callers will block).

Messages sent to a port via send rights are delivered reliably (reception of a message guarantees that all previous messages were received) and they are received in the order in which they were sent by any given task (or any given kernel). Messages sent to a port via send-once rights are unordered.

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.

Basic Manipulation

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_nam-*

e_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.

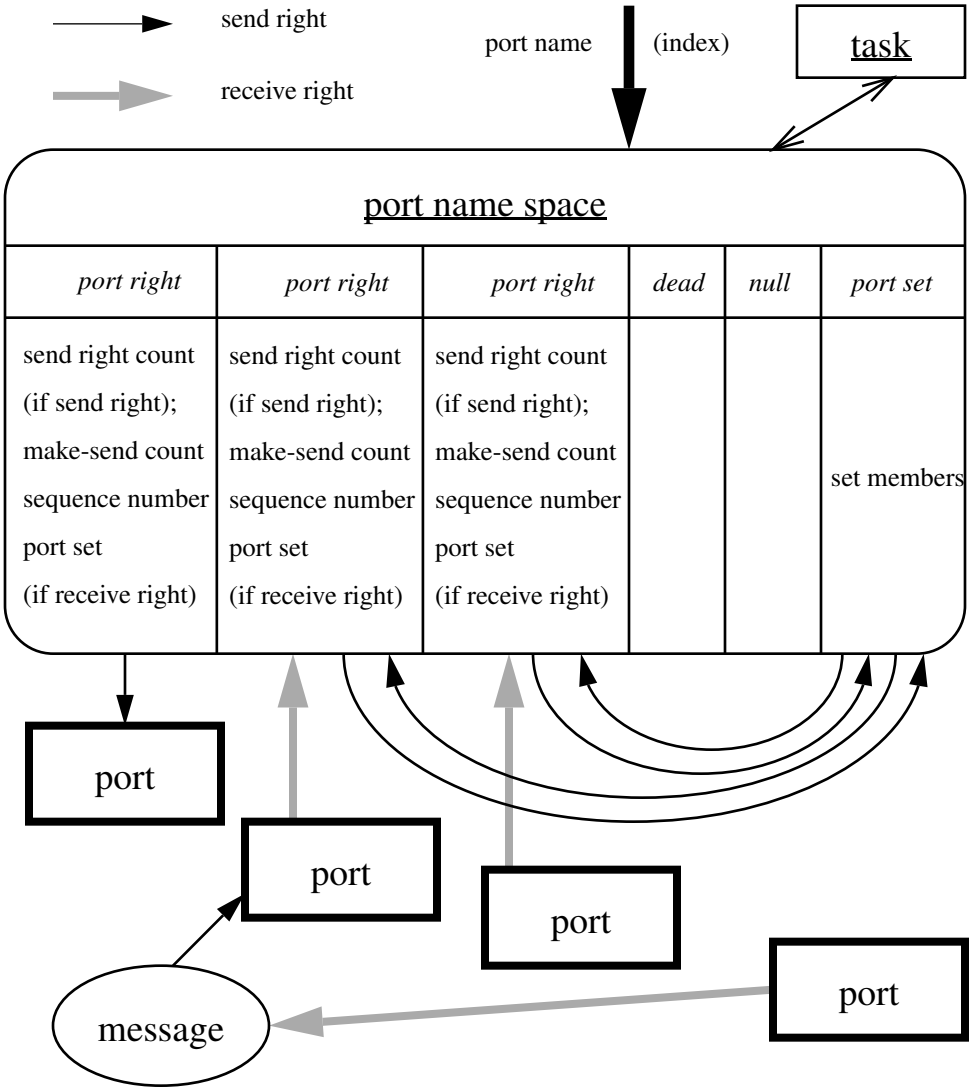

FIGURE 4 **Port Right Structures**

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* [*port_name*] → **mach_port_allocate** and *port_name_space* [*port_name*] → **mach_por-**

t_allocate_name. 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_attributes**.

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

Send-Once Rights

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 two 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 or the receive right moved. Since there will be no forth coming no-senders notification, a send-once notification is generated instead.
- In the process of performing a 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 (refer to the details of message transmission below). 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.

No-More-Senders Notification

The system tracks the send 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 of the receive right cancels the outstanding no-more-senders notification request which, by virtue of the destruction of the send-once right that would have been used for the no-more-senders notification, sends a send-once notification to indicate this cancelation. Note that the no-more-senders notification indicates the absence of send rights; there may well still be outstanding send-once rights.

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**. Generating 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 request 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_attributes**.

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 a 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 at the time the notification was generated; there may still be outstanding send-once rights. A task can keep track of the number of outstanding 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.) 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* trailer 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**.

Manipulation 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.

Message Send

The send operation queues a message to a port. The message carries a logical copy of the caller's data. The caller can freely modify the message buffer after the send operation returns without affecting the data sent.

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. (Kernel generated messages also forcibly queue.)
- 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 sending of a message is a two-step process. The first step involves constructing a kernel copy of the message, the second, the queuing of the message. Failures during the first step (invalid port rights or data addresses, for example) cause the message send to fail (with an error return) with no ill effect. Failures during the second step (the send time-

out value is exceeded or an interrupt (**thread_abort**) occurred) also cause the send to fail, but 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 flag.

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. Note that the kernel does not guarantee that the out-of-line memory ranges have the same addresses as before nor that the port rights have the same names as before.

Message Receive

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 body will be copied. The format of the received message is the same as when sent. If the message does not fit, it is normally destroyed. An option (MACH_RCV_LARGE) allows the caller to receive an error instead, along with the buffer size (or out-of-line sizes) 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 reception of port rights follows the same rules as the insertion of a port right by any other means as described under port name spaces. The port right descriptors describe the type of rights received.

The receive operation is also a two step process: de-queue the message and then make a copy in the receiver. The first step may fail because the specified receive time-out value was exceeded or because the receive was aborted (**thread_abort**). These situations do not affect the message that would have been received. Most failures occur during the second step causing the message to be destroyed.

There is a notification that can be requested as a result of a **mach_msg** call. The 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 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 when aborting the service request would be important. However, in the typical case in which the client does not die, the receiver of the original request will eventually send a reply into the reply right, which, (typically) being a send-once right, will destroy the reply right and generate an otherwise meaningless port-deleted notification. An optimization to cancel the dead-name notification request automatically when sending this reply is provided by the MACH_SEND_CANCEL option to **mach_msg**.

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

Kernel Messages

Messages sent from the kernel (including reply messages) are normal messages with no special properties (other than originating from the kernel). Messages sent to the kernel, however, are specially handled.

When a message is sent to the kernel (an invocation is made of a kernel object), in general, it is the sending thread, in kernel context, that performs the work. Thus, even if the kernel operation is requested asynchronously (the request sent without explicitly waiting for the reply), the operation is still synchronously performed. The send operation may or may not return until the operation is complete.

Out-of-Line Memory Transmission

Out-of-line data is logically copied between the sender and the receiver. The nature of this copy and the mechanisms involved depend on a variety of factors.

The sender of an out-of-line region can specify that the kernel make a “physical” copy of the region. This is the most efficient choice for small to moderate sized regions. This choice also guarantees that the sender and receiver will have prompt and safe access to their copies (in contrast to a virtual copy). Out-of-line port arrays are always physically copied.

The disadvantage of the physical copy option is that the kernel must really copy the data (at the time the message is sent). The other memory transmission mechanisms permit the copy to be delayed (possibly indefinitely) and are therefore preferred for large data, or in special circumstances.

Requesting a “virtual” copy permits the kernel to use its virtual copy (copy-on-write) mechanisms to efficiently copy large amounts of data (refer to CHAPTER 5 for details). The data is passed as a virtual copy (the kernel constructs a virtual memory image of the set of pages that define the region). If the kernel constructs a virtual copy, it does not transmit the portion of the first page preceding the data in the virtual copy and the portion of the last page following the data in the virtual copy, these portions will appear as zero if the receiver specifies dynamic allocation.

The disadvantage of a virtual copy is that the copy-on-write mechanisms are fairly expensive, and cause both the sender and receiver to experience indeterminacy in their access to the data. Also, the receiver may see a copy that is effectively backed by the sender’s memory manager, possibly an integrity concern for the receiver.

The third set of copy mechanisms provides a “buffered” copy. With a buffered copy, the copy is not logically made at the time the data is sent, but at some (undefined) time between when the message is sent and some “boundary” event. (It is quite possible that the data is never actually copied depending on the nature of the boundary event and the sender’s and receiver’s access to the data.) Prior to the point (if there is one) at which the

copy is made, changes to the sender's data will be visible by the receiver; also, since the receiver receives a writable region, the receiver's changes will be visible as well (i.e., the sender and receiver share access to the data). Because of the difficulty of specifying such boundary events in a generic way (which would have to be known to the sender, receiver and transmitting kernel), buffered copies are currently reserved for use by the kernel.

The purpose of the buffered copy is to permit local case optimizations in certain data transmissions between tasks and the kernel. (They are local case optimizations because, when crossing node boundaries, the data must be copied. This is why this memory transmission scheme is not defined as "shared" access; the data is only shared while it can be; in other cases, the data is actually copied.)

In general, all memory transmissions involving synchronous kernel interactions (those completed before the kernel message send returns) use buffered copies. But, since the interactions are synchronous, the sender cannot detect this sharing. In general, asynchronous kernel interactions use physical or virtual copies (to get real copy semantics), with few exceptions. Currently, the only exception is the "clean-in-place" mechanism in which the data sent in the **memory_object_data_return** message is a buffered copy (and **memory_object_data_return_completed** defines the boundary point).

For each out-of-line region transmitted, the kernel will return a descriptor in the received message body locating the memory as well as indicating whether the kernel chose to (or the sender directed to) send a physical copy (thereby guaranteeing deterministic behavior) or send a virtual (or buffered) copy. Although backed by the default memory manager, the kernel's virtual copy optimizations may use the page images already in memory or may fetch page images from the memory manager that backed the sender's memory. The optimal case for a virtual copy is when the sender has used the de-allocate option.

There are two options for the reception of an out-of-line memory region (this includes data regions as well as the port names in an "out-of-line" port array). In the default case, the received region is dynamically allocated as temporary memory in the receiver's address space (as if by **vm_allocate**). If the kernel transmitted a virtual copy, the received data will appear in allocated space at the same alignment within a page as that of the sender. In all other cases the allocated data will start at the beginning of a page boundary. Under no circumstances will the reception of a message with this option stall the receiver by inadvertently referencing the virtual copy itself.

The alternate out-of-line reception mode causes the out-of-line region to overwrite the bytes in a specified area (as if by **vm_write**). If the region does not fit in the specified area, the receiver sees an error. Under various circumstances depending on the method of data transmission and data alignments this option may perform page manipulations or may actually copy data. This operation may possibly interact with the memory managers backing any virtual copies. This may arbitrarily stall the receiver.

The choice of the two reception modes depends on the MACH_RCV_OVERWRITE option. When not set, all received regions will be dynamically allocated. If set, the receive buffer is considered to describe a "scatter" descriptor list. This list contains out-of-line memory and port array descriptors, of the same form as appear in sent messages, that the kernel will scan to determine what to do with each received region. If not enough descrip-

tors are supplied to handle the sent number of regions, any additional regions are dynamically allocated. The kernel looks at each out-of-line descriptor to determine the disposition of the corresponding region—dynamic allocation or overwrite (with the address and size fields of the descriptor used to locate the region to be overwritten, an insufficient size causing an error return).

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 any thread executing within the task owning the virtual address space is allowed to reference. A virtual address space is named by its owning task.

Creation and Destruction

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.

Memory Regions

A virtual address space consists of a sparsely populated indexed set of memory pages. (The size of a page can be found with *host_name* → **host_page_size**.) 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.

Mapped Memory Objects

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*], *memory_object_representative* → **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 memory object representative port that implies the abstract memory object that backs the storage for the range as well as the allowed access modes to the memory object. A null port specifies that the system's default memory manager is

to be used. The default memory manager provides initially zero-filled storage that is paged against the system's paging space and which will not persist between system bootloads. *virtual_address_space* [*memory_range*] → **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**.

Address Space Control

Each page in an address space has a variety of attributes that govern the legitimacy or behavior of accesses to the page.

- *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 via the memory object representative; 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_machine_attribute** — Set machine specific hardware properties for the memory range.
- *virtual_address_space* [*memory_range*] → **vm_wire** — "Wire" (force to be and stay resident) a range of memory. Wired memory is counted against the task's wired kernel memory limit. The kernel maintains a count of the number of times a region is wired and does not un-wire it until the count returns to zero.
- *virtual_address_space* [*memory_range*] → **vm_behavior_set** — Specify the expected usage pattern (normal, random, sequential, reverse sequential) for the range as a hint to guide the kernel's page replacement algorithms.

Memory Access

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_read** — Copy out a memory range.
- *virtual_address_space* [*memory_range*] → **vm_write** — Copy in a memory range.
- *virtual_address_space* [*memory_range*] → **vm_copy** — Copy a memory range from one place in a virtual address space to another.
- *virtual_address_space* [*memory_range*], *virtual_address_space* [*memory_range*] → **vm_remap** — Maps the objects underlying a given memory range of some task. The effect is as if the mapping task performed a **vm_map** call given the memory object representative used by the mapped task for each object in the mapped task's range with the same access modes and inheritance. Since this call maps the underlying objects and not the mapped task's address space, changes to the mapped task's address space (terminating objects, as well as some manipulations of those objects) may result in the memory the mapping task references to no longer be the same memory as in the mapped task.

- *virtual_address_space* [*memory_range*] → **vm_msync** — Force flush (or clean) a range of virtual memory to backing storage (its memory manager).

Reserved Memory

Not all memory in an address space is allocated exactly where the task may desire. Randomly allocated memory (an option to **vm_allocate** or **vm_map**) or dynamically allocated out-of-line memory received in IPC messages appears at random addresses. In some environments it is important to identify the areas in which such allocation must not occur to avoid conflicts with other expected uses of that memory (such as future growth of the stack). For this purpose, such memory can be marked as *reserved*, or protected from random allocation, with *virtual_address_space* [*memory_range*] → **vm_reserve.** *virtual_address_space* [*memory_range*] → **vm_unreserve** cancels such a reservation. Reservation is an orthogonal property of a memory range; even if an object is allocated in reserved memory (via explicit allocation), the region continues to be marked as reserved and will be reserved when that object is de-allocated. Inheritance can be set for reserved regions and reserved regions can be inherited.

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 memory object representative port to name the memory object that backs that range (*virtual_address_space* [*memory_range*], *memory_object_representative* → **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 disposal.
- The client task dispels 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 kernel's messages are sent to the memory manager via its abstract memory object port; messages to the kernel are via the kernel's memory cache control port. 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 memory object representative port specified in the client's **vm_map** call implies (in a way to be described) the abstract memory object port that 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 can also generate 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.

FIGURE 5 shows the client visible virtual memory structures. Shown are three memory ranges, 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 two memory object representatives (representing read and read/write access) and the memory manager task.

A reserved but not allocated region (not shown) would be marked with only the reservation flag and an inheritance attribute; no other attributes apply.

Memory Object Initialization

The result of memory object initialization is to form the binding between the memory manager's abstract memory object (backing store) and the kernel's memory cache object (memory image). The memory manager needs to communicate to the kernel the abstract memory object port over which the kernel will send subsequent manipulation messages and the kernel must communicate to the memory manager the memory cache control port over which the manager will send subsequent manipulation messages.

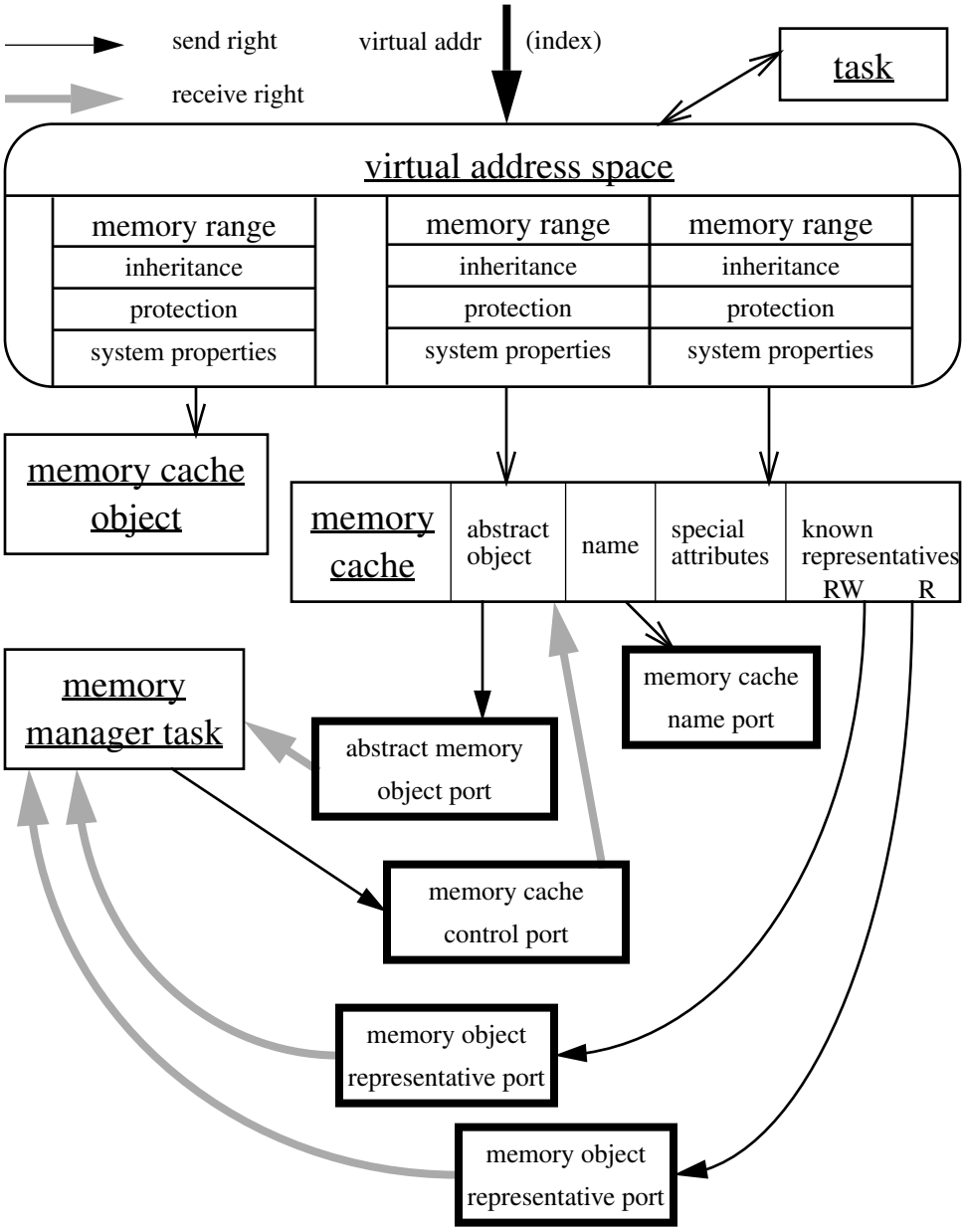

FIGURE 5 **Virtual Memory Structures**

Since the abstract memory object port is the port over which the kernel will send messages according to specially designed protocols, it is not a good idea for clients to also have access to this port as they may confuse the memory manager. Indeed, there are invalid sequences of messages that a client could send over the abstract memory object port for which the memory manager has insufficient knowledge to protect itself. For this reason, and for the sake of access control for memory objects, memory managers typically do not distribute their abstract memory object ports except to the kernel; instead, clients are given send rights to memory object representative ports which in turn imply the associat-

ed abstract memory object port and associated set of allowed access modes to the associated memory object.

Both abstract memory object ports and memory object representative ports are normal Mach ports (that is, their receiver is a Mach task instead of the kernel) so they possess their special properties by the manner in which they are used.

When a client specifies a memory object representative port to **vm_map** to a kernel that does not yet know the identity of the object so represented, the kernel will engage in a special dialog using the representative port. If the kernel does know the identity of the associated object, the **vm_map** call can be executed directly with no interaction with the memory manager.

The **vm_map** call causes the kernel to send *memory_object_representative* (*host_name*) → **memory_object_notify**, which includes a send right to a memory cache control port. The memory cache control port is assigned default attributes at this time. (This port should not be considered to be fully established and its attributes properly set until after the **memory_object_establish** message is sent.)

Since a fake **memory_object_notify** message could have been generated by a client holding the representative port (instead of by the kernel as a result of **vm_map**), the memory manager does not directly reply to this message but instead sends its response message to the host name port supplied in the notify message (which it can verify really is a host name port either by checking **mach_host_self** or querying a name server). After verifying that the host name port is valid, the memory manager responds with *host_name* (*ledger*, *memory_cache_control*) → **memory_object_establish** specifying the abstract memory object port and the object attributes, as well as the accesses allowed by the memory object representative (as well as a restricted identity, if any, that tasks attempting to map via this representative must have). It is permissible that the memory object representative port be specified as the abstract memory object port, although this introduces a potentially vulnerability for the manager. The object attributes allow the specification of the copy strategy (discussed below) as well as other behavioral attributes. These attributes may later be changed with *memory_cache_control* → **memory_object_change_attributes** and examined with *memory_cache_control* → **memory_object_get_attributes**.

Note that the manager includes the memory cache control port in this message. Since the control port was created specifically for this use of the representative port, it has not been given away to any other task and so it can be used by the manager to authenticate itself to the kernel (that is, that the manager is the receiver of the **memory_object_notify** message).

If the kernel completes the initialization, no direct reply is sent. At this point the memory manager must be ready to respond to requests on behalf of this memory object.

However, if the kernel was not expecting the **memory_object_establish** message (a mischievous client forged the original **memory_object_notify** message or the kernel decided to terminate object initialization), the kernel responds with *abstract_memory_object* → **memory_object_rejected**. This informs the manager that the port it thinks is a memory cache control port really isn't. Note that in such a case the memory manager did not di-

vulge any information to the client; it merely sent an extraneous message to some kernel that didn't expect (or accept) it.

This sequence involving the host name port and a memory cache control port authenticates the kernel to the memory manager and vice versa. It depends on the ability of the memory manager to authenticate the name port given to it. To be able to interpose on the kernel to manager dialog, the manager would not be presented with the host name port by the intermediary but some other "name" port. For this to work, the manager must either not authenticate the name port given to it or the authentication service it uses must (possibly by interposition) vouch for the authenticity of the name port given to the manager.

The above sequence concerns the initialization of a memory object not yet known to the kernel. It is invoked whenever the kernel is presented (via **vm_map**) with a memory object representative port it does not know. Note that the memory object underlying the representative may well be known to the kernel; the kernel just doesn't know the association and so the kernel must assume that this representative implies the initialization of a new memory object. If the memory object is initialized, for subsequent **memory_object_notify** messages (from the same kernel), the memory manager responds with *host_name* (*memory_cache_control*) → **memory_object_representative**. This call only supplies the abstract memory object port and allowed access modes for this representative (as well as a restricted identity, if any, that tasks attempting to map via this representative must have). When the kernel receives this message, it will destroy the (redundant and useless) memory cache control port supplied. The MIG stub for **memory_object_representative** moves the send right as an optimization so that the manager need not explicitly discard its right. It is possible, though, that this message will race with memory object termination (the manager believes the memory object to be initialized but it is no longer). In such a case, the kernel will discard the **memory_object_representative** message (since the object is not initialized from the kernel's point of view). The manager will detect this situation by the presence of a **memory_object_terminate** message on the abstract memory object port. When the kernel discards the **memory_object_representative** message, it re-sends the **memory_object_notify** message. The manager should reply once again with **memory_object_establish**. As a general rule, unless the manager knows for sure that a memory object is un-initialized (the object never was initialized or the manager saw a **memory_object_terminate** message), the manager should respond to **memory_object_notify** with **memory_object_representative**. If the kernel was not expecting the **memory_object_representative** message (a client fabricated a **memory_object_notify** message), the kernel simply ignores it. The fact that the manager sent this message means that the manager believes the memory object to be initialized and so the **memory_object_notify** message to which the manager was responding provided no information to the manager. The kernel is always free to forget a representative to abstract object association at any time; the kernel's ignoring the **memory_object_representative** message is the same as if the kernel accepted it and then immediately forget the association.

The memory cache object can also have a name port (provided to clients through **vm_region**); the manager can obtain this with **memory_object_get_name**.

Basic Page Manipulation

The kernel requests data for memory with *abstract_memory_object* → **memory_object_data_request**. Typically, the kernel requests (as well as supplies) only a single page at a time. The kernel reserves for itself the right to operate on multiple pages at a time (biased by the memory region's set behavior). The memory manager can set a *cluster size* for a memory object specifying the number of pages preferred for data return transfers. Various factors affect how many pages are actually transferred at one time.

The memory manager supplies the requested data with *memory_cache_control* → **memory_object_data_supply**. This call also supplies the maximum allowed accesses for the data. If the memory manager cannot supply the data because of some error, it responds with *memory_cache_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_control* → **memory_object_data_unavailable**. In this case, the kernel supplies the missing data (zeroes, or a copy of data in the case where the kernel was performing a virtual object copy (discussed below)). Memory managers are not permitted to supply pages not requested by the kernel.

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**. In this process, the kernel sends a virtual copy of the pages (effectively converting the physical memory pages from being resident pages associated with the memory object to being pages associated with a default memory managed temporary object). In this way, if the memory manager cannot move these pages to their destination in a reasonable time, the kernel can evict these "temporary" 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.

Once the pages are evicted, the kernel is free to request them again at any time, possibly immediately. As such, a memory manager (especially one that is multi-threaded) must protect itself against back-to-back data return and data request messages for the same pages. (Refer to the *Server Writer's Guide* for a discussion of this situation.)

Clean-in-Place

The kernel is confronted with several problems when returning (evicting, cleaning or flushing) pages to a memory manager.

- The kernel must be prepared to evict these pages to temporary backing storage if the manager does not remove them itself in a reasonable time.
- The kernel makes eviction decisions concerning single pages. For efficiency, though, the kernel would like to return (but not evict) multiple pages to a manager (even if the others would not otherwise be considered at this time) so that the manager can update multiple pages in its backing storage at once. These additional pages want to remain accessible to their mapping tasks.

- In principle, the manager can modify the pages returned to it (they appear as normal out-of-line memory) and so the kernel must be prepared to make writable copies of the pages.

The kernel does its best to optimize the page return mechanisms. The greatest performance is achieved when the manager will not modify the pages and can be trusted to dispose of them. Special high performance mechanisms (internally called *clean-in-place*) are used by the kernel if the following conditions are true:

- The memory manager is privileged with respect to memory resource consumption and asserts this privilege by sending a **memory_object_establish_priv** message to the host control port instead of the host name port.
- The memory manager is on the same host (node) as the memory cache object.
- The memory manager does not modify the pages.

The use of these special mechanisms affects the semantics of the data return in that the manager must explicitly indicate to the kernel when it has disposed of the pages. The manager does this by sending a *memory_cache_control* → **memory_object_data_return_completed** message. The pages of memory will not be freed until the kernel receives this message. As a result, no subsequent data request message will be seen by the manager until it disposes of the returned pages.

Memory Object Termination

When all tasks remove their mappings for a memory object representative, the kernel discards its association of that representative to the underlying abstract memory object (c.f. object cacheability below). The memory manager is not notified. The memory manager must be prepared to receive a **memory_object_notify** message for a representative at any time. Each subsequent **memory_object_notify** message will include a unique memory cache control port.

When all tasks remove their mappings for the underlying abstract 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 the *temporary* attribute) prior to actually terminating the memory cache object (and sending the terminate message). The terminate message sends the receive right for the memory cache control port to the manager so that the manager can obtain any messages it had sent to that port after the kernel terminated the object but before the manager received the terminate message.

Alternatively, the memory manager can call *memory_cache_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. Terminating a memory object representative port does not terminate the underlying memory object, but it does invalidate all mappings obtained through that representative.

One of the object attributes set when the memory object is initialized by **memory_object_establish**, or thereafter altered by **memory_object_change_attributes**, is the "object cache" attribute. With this attribute set, instead of dissolving the representative–abstract object association, or 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_notify** 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; if the object is not terminated, a **memory_object_change_completed** message will be sent, otherwise only a **memory_object_terminate** message will be sent.

Synchronization with Backing Storage

A client of a memory object often desires to know that changes made to the memory object are reflected on backing storage. The client makes such a request by executing **vm_msync** against a memory range. The basic response of the kernel is to clean or to flush (depending on a flag given by the client to **vm_msync** and on the value of the memory object's *invalidate* attribute) the set of pages to the manager and to engage in a synchronization dialog with the manager.

After cleaning or flushing the range, the kernel sends *abstract_memory_object* → **memory_object_synchronize** specifying the memory object range as well as the client's flags. One of these flags is an *asynchronous* flag. If set, the client is only interested in having the pages queued to backing storage; if not set, the client is interested in the pages actually being on backing storage. When the manager has achieved the desired effect, it responds with *memory_cache_control* → **memory_object_synchronize_completed**. The client returns from its **vm_msync** call at that time. The range specified to **memory_object_synchronize_completed** must match that of the corresponding **memory_object_synchronize** message. Although the kernel may have multiple synchronize requests outstanding at a time, it will never have overlapping requests outstanding.

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 available 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_port* → **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_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_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 (such as a *fsync* operation). 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, the following state variables 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 may want more access than it currently has.
- If the memory manager is using the "clean-in-place" mechanisms, the kernel will not request a page it has written until a **memory_object_data_return_completed** response has been received. This state affects the disposition of pages and delays eventual subsequent page-in. It is not shown in the transition diagrams.

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:

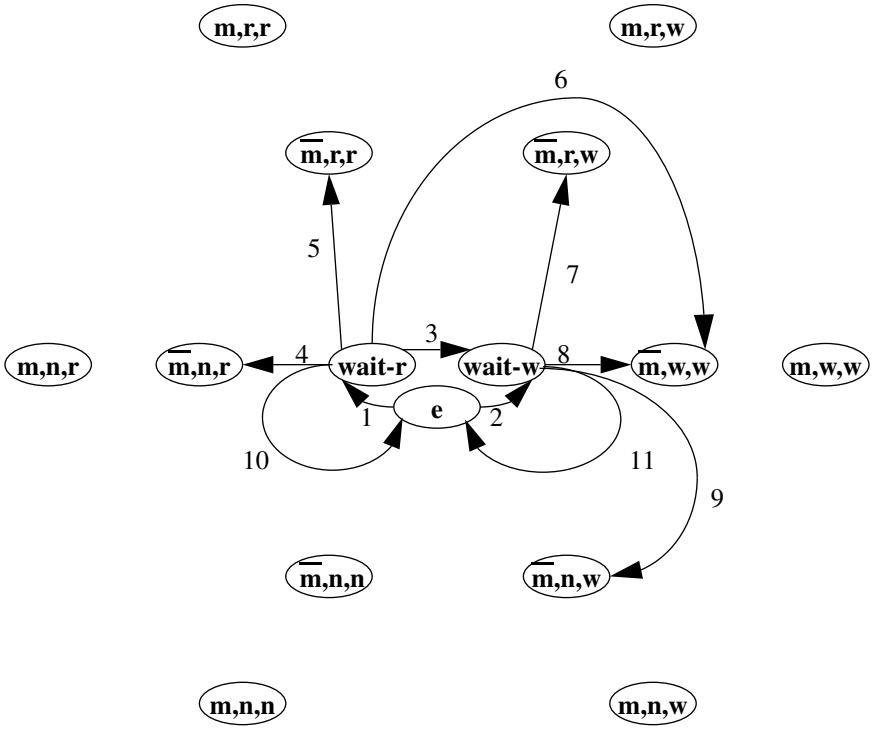

FIGURE 6 **Page-in Transitions**

- 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 has the kernel fabricate a page (**memory_object_data_unavailable**), specifying no access. The kernel will send a **memory_object_data_unlock** message requesting read access.
- 5—The manager supplies the page, or has the kernel fabricate a page, with read access. The kernel simply accepts it.
- 6—The manager supplies the page, or has the kernel fabricate a page, with write access. The kernel accepts the page, marking it as writable.
- 7—The manager supplies the page, or has the kernel fabricate a page, with read access. The kernel will send a **memory_object_data_unlock** message requesting write access.
- 8—The manager supplies the page, or has the kernel fabricate a page, with write access. The kernel simply accepts it.
- 9—The manager supplies the page, or has the kernel fabricate a 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* with or without *clean* transitions (**memory_object_lock_request**) when the page is in an un-modified state.
- FIGURE 8 shows the *lock* without *clean* transitions for initially modified states.
- FIGURE 9 shows the *lock* with *clean* transitions for initially 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 transition 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. Notice that FIGURE 7 and FIGURE 8 are effectively the same. What these transitions show is that, in the absence of the *clean* operation, the page maintains its modification status (if the page was initially unmodified, specifying clean will not affect the modification status); the only change is that the current accesses are changed to those specified in the lock request, and the desired accesses are also set to these same values.

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**) prior to the **memory_object_lock_completed** message. As such, FIGURE 9 differs from FIGURE 7 and FIGURE 8 in that the *clean* operation removes the modified flag from the page.

FIGURE 10 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.

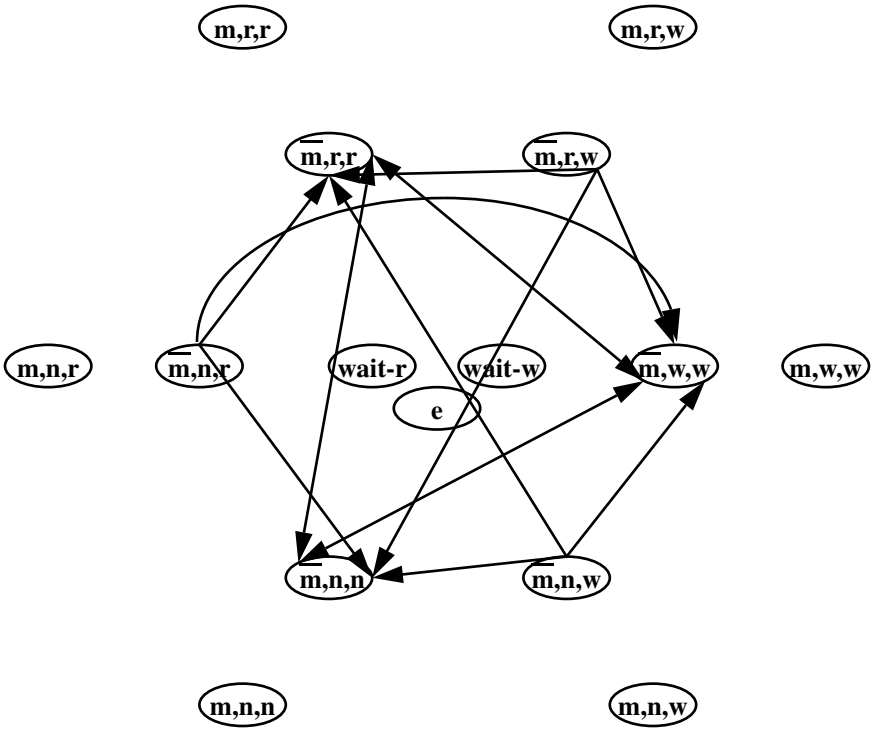

FIGURE 7 **Lock, with or without Clean (un-modified) Page Transitions**

- 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** 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.

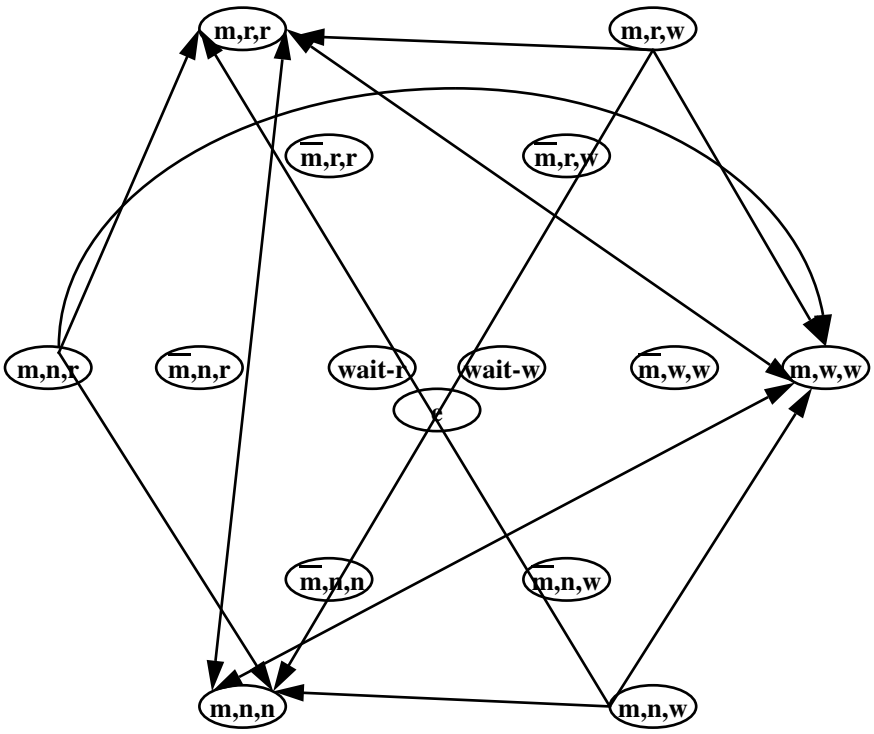

FIGURE 8 **Lock, without Clean (modified) Page Transitions**

Virtual Copy Optimizations

There are three 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 **vm_copy** copies one portion of an address space to another portion of that space.
- When a memory range is passed as a virtual out-of-line region in a Mach message. (This includes **vm_read** and **vm_write** operations.)

These situations occur 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.

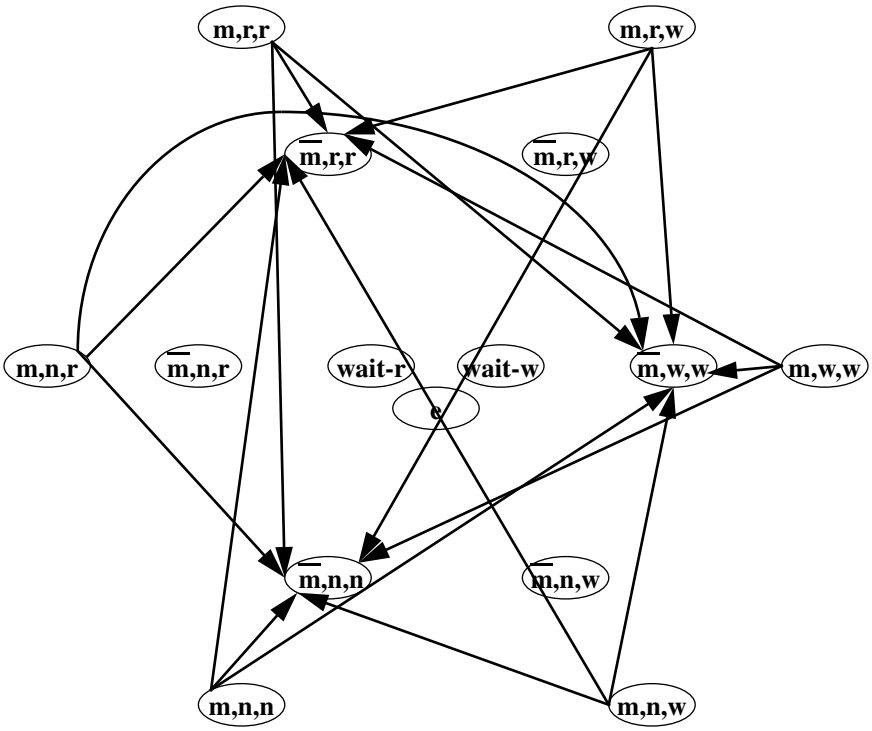

FIGURE 9 **Lock, with Clean (modified) Page Transitions**

- 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.

The memory object resulting from a virtual copy is always an object managed by the default memory manager. (Even in the "page stealing" case mentioned above, the copy object is still a default memory manager object; only the resident pages are stolen, not the original memory object.)

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).

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.

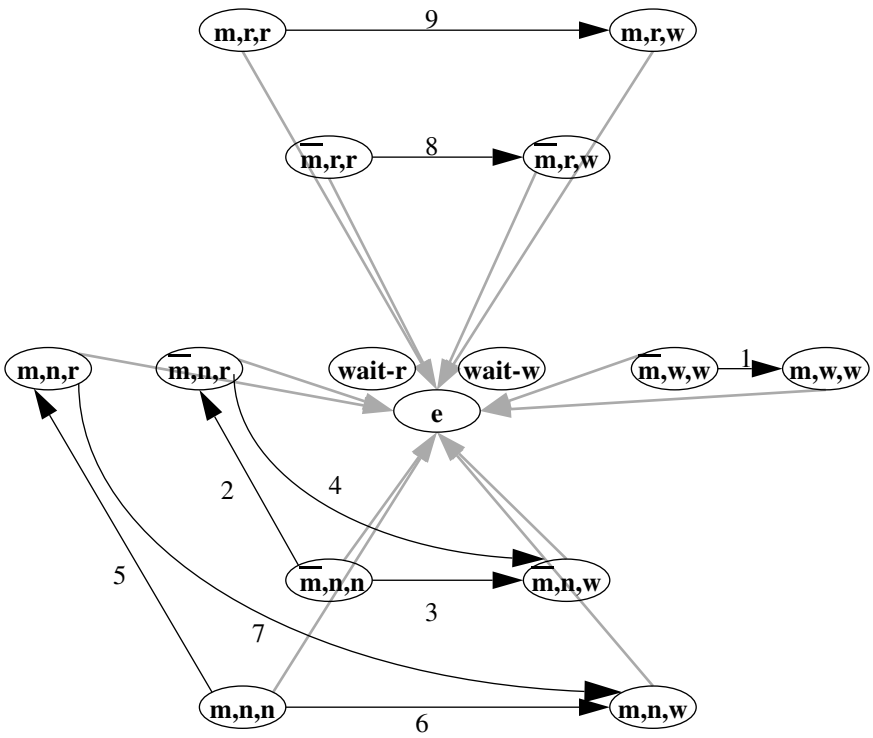

FIGURE 10 **Peek, Poke, Evict, Flush Page Transitions**

Referencing the virtual copy may cause pages to be obtained from the original memory object (those that had not yet been copied into the copy), thereby interacting with that object's memory manager. As such, the behavior of the referencing task may be affected, especially if the original memory manager does not respond to the kernel's page requests.

Virtual Copy Dialog

When a range of memory is to be copied, the kernel creates a new abstract memory and 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_establish** when the object is initialized) for the old memory object, as discussed below.

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.

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 an alternate copy strategy is 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 disable this optimization, the MEMORY_OBJECT_COPY_NONE strategy is provided. 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 that time) from the old memory object (thus requesting all pages at this time from the memory manager).

The use of the MEMORY_OBJECT_COPY_NONE strategy can adversely affect the performance of memory management because of the aggressive copies performed. If a memory object will be manipulated in ways that would invalidate the memory cache but the memory manager has complete knowledge and control over when these manipulations occur, the memory manager may still use the MEMORY_OBJECT_COPY_DELAY strategy with the help of the force copy option to the **memory_object_lock_request** call. This option would be used by a memory manager when it realizes it is about to invalidate the kernel's cache. This option causes the kernel to aggressively evaluate all delayed copies of the specified region of the memory object. (Note that this may entail kernel requests for pages from the manager; the **memory_object_lock_completed** message will not appear until these page requests are satisfied.)

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, etc.). 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** (c.f. **norma_set_special_port**). This is initially the kernel supplied default 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 *default_memory_manager* → **memory_object_create**. A memory object to be managed by the default pager can be created by the privileged *default_memory_manager* → **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 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_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 or files as backing storage. The set of partitions to be used can be dynamically configured with *default_memory_manager* → **default_pager_paging_file**. The amount of total space and free space available can be found with *default_memory_manager* → **default_pager_info**.

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 Kernel 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. Clients have no control over this algorithm (with the exception of the **vm_wire** call). 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 (except for the *clean-in-place* mechanism reserved for "trusted" managers).

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.

Ledgers

Some Mach kernel resources are “created” at system initialization (such as clocks) and therefore need no resource accounting. The ability to create kernel objects (and therefore consume kernel resources) dynamically can lead to the ability to consume excess resources thereby leading to denial of system service. Resource *ledgers* provide the mechanism to limit such excess consumption.

A resource ledger contains a limit and the current usage count for a given resource. Each ledger limits a particular type of kernel resource:

- Amount of wired kernel memory (data structures, wired pages)
- Amount of (temporary) pageable memory (memory backed by the default memory manager).

With the exception of tasks and threads, every dynamically created, first-class (port named) kernel object consumes wired kernel memory and “draws” that kernel memory from a ledger specified when the object was created. A ledger reference is required in the **memory_object_establish**, **processor_set_create** and **device_open** calls. (**clock_alarm** also needs a ledger from which the alarm structure is allocated.) These objects (memory cache objects, processor sets and devices) only consume kernel wired memory; they do not consume default memory manager space (any out-of-line memory sent to them is either wired in the process or immediately used and discarded).

Tasks (and their logical contents) consume all of the various types of kernel resources (kernel wired memory and default memory manager space). When a task is created (**task_create** or the NORMA equivalents), a set of ledgers, one for each pool of kernel managed resources, must be supplied. (If they are not, the corresponding ledger from the parent task is used.) The resources consumed by threads are considered to be associated with their containing task.

Non-first-class objects are drawn from the object that “owns” them. For example, a right in a task’s port name space is counted against the task. A port (including the queued messages) is counted against the task holding the port’s receive right. A receive right in transit is counted against the transferring port (which means the task holding that port’s receive right).

When a first-class object is created, the creation message must carry a (send) right to one or more ledgers. A ledger represents resource consumption on a single kernel. If the supplied right indeed names a ledger on the kernel creating the object, that ledger is used as the source of the resources. However, if the right is anything other than a local ledger, the kernel sends a **ledger_get_remote** message to that right, which includes the kernel’s host name port. If the response contains a local ledger port, the kernel uses it, otherwise it sends **ledger_get_remote** to that right. This mechanism supports distributed resource ledgers in the following way:

- With **ledger_set_remote**, a ledger is assigned a “remote ledger service port”.
- This ledger is used as the ledger for a create operation. If the ledger is local to the target kernel, all is fine.

- For a non-local creation, the target kernel sees that the supplied ledger is not a local ledger. The kernel sends a **ledger_get_remote** message to it, including the host name.
- The (remote) ledger receives this message, ignores the host name and returns the remote ledger service port.
- Assuming that the remote ledger service port is not a local ledger, the kernel sends a **ledger_get_remote** message to this service port.
- A server holding the receive right on the remote ledger service port uses the identity of the service port as well as the host name of the target kernel to locate or create a suitable ledger on that kernel.
- The port for a ledger on the target kernel is sent to that kernel and used.

When a ledger is destroyed (**ledger_terminate**), all objects drawn from it are destroyed.

The current limit and balance for a ledger can be found with **ledger_read**.

A ledger is created from a parent ledger with **ledger_create**. This creates a subordinate ledger (which is drawn from the parent). At creation time, some of the parent's resources (some of its limit representing its ability to consume resources) can be transferred to the child. Resource limits can be transferred back and forth between a parent ledger and a child with **ledger_transfer**. Note that the ability to consume resources is transferred — a check to determine if an allocation will exceed a ledger examines only the ledger from which the object is directly drawn, not any of that ledger's parents.

When a host (node) boots, the kernel creates a *root* ledger of each ledger type, supplying these ledger ports to the bootstrap task. These ledgers have a special limit of LEDGER_ITEM_INFINITY, which permits any amount (even infinite) to be withdrawn without decreasing the ledger limit.

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**. The list of hardware processor slot numbers is obtained by *host_name* → **host_processor_slots**.

FIGURE 11 shows the client visible processor and processor set structures. The processor set control port provides access to the contained tasks and threads. The set of all processors is found via the host control port.

The following operations exist:

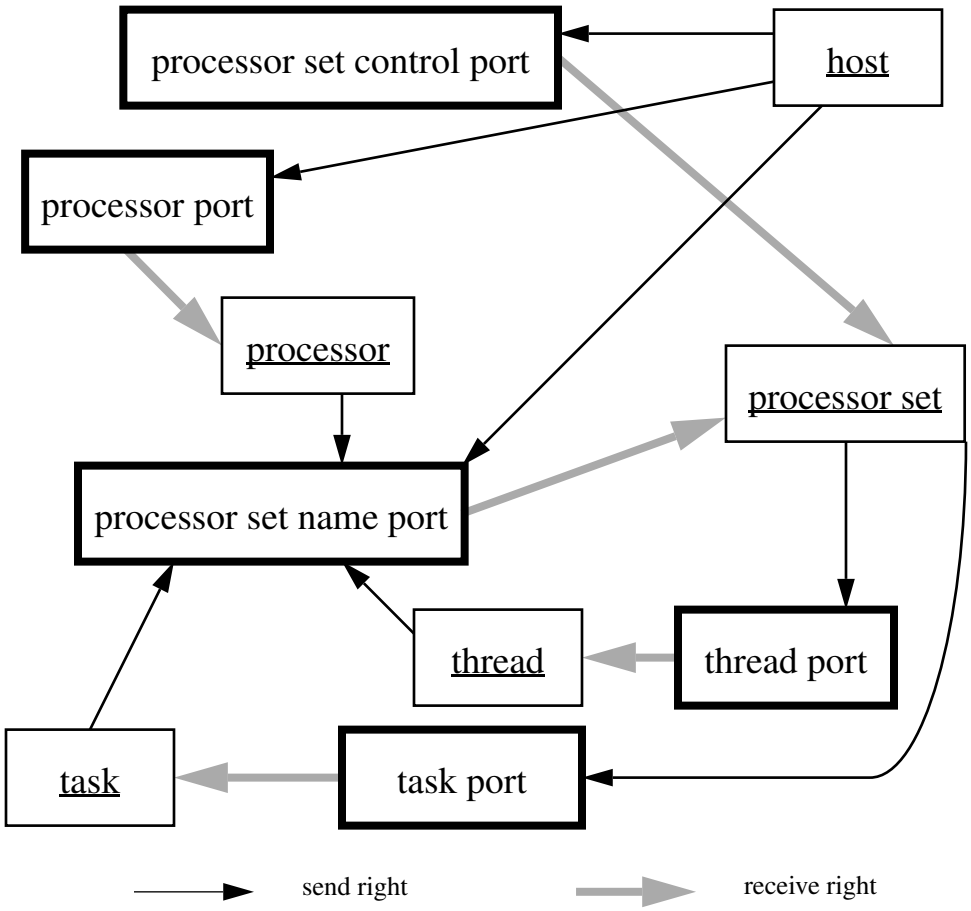

FIGURE 11 **Processor Structures**

- *processor*, *processor_set_control* → **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 contains a set of processor sets. There is always at least a default processor set. 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_control* → **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 *processor_set_name* (*host_control*) → **host_processor_set_priv** call which translates a processor set name port into its control port.

All processor sets other than the default set must be created explicitly, with *host_name* (*ledger*) → **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 reassigned to the default processor set.

A processor is added to a processor set with *processor*, *processor_set_control* → **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. Also, children of a task inherit the parent's processor set association.

A task is assigned to a processor set with *task*, *processor_set_control* → **task_assign**, a thread with *thread*, *processor_set_control* → **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.

This kernel internal task is 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 this task as the first task 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, for example, a base priority for the timeshare policy.

Any given processor set supports the scheduling of its threads according to the scheduling policies implemented for that processor set. Each thread has an associated scheduling policy which it inherits from its containing task. The privileged call *task* (*processor_set_control*) → **task_set_policy** allows a task to have assigned to it any policy implemented by its assigned processor set. This policy can be viewed with **task_info**. Changing the policy for a task does not affect any contained threads unless an option to the set policy call is used. The privileged call *thread* (*processor_set_control*) → **thread_set_policy** allows a thread to have assigned to it any scheduling policy implemented by its assigned processor set. This policy can be viewed by **thread_info**. Since some of the implemented scheduling policies permit threads to overly consume processor time, a subset of the implemented policies may be enabled for random tasks/threads with *processor_set_control* → **processor_set_policy_control**. The currently enabled set can be found with *processor_set_name* → **processor_set_info**. A task (without privilege) may select a policy from this subset of enabled policies with *task* → **task_policy**, a thread with *thread* → **thread_policy**. When a task's/thread's processor set assignment is changed, it maintains its policy unless the new processor set does not have it enabled, in which case the task's/thread's policy is set to the processor set's default policy (*processor_set_control* → **processor_set_policy_control**). However, disabling a policy for a processor set does not affect any tasks/threads currently in that set (unless an option to the **processor_set_policy_control** call is used).

For a given policy, a task/thread has a base set of scheduling attributes as well as limits on the values it may choose for its base attributes. (The active attributes are called the *base* set because some policies, timeshare for example, maintain a dynamic (current) set derived from the base set.)

When a task/thread is created, it inherits the scheduling limits of its parent. When a task/thread has its processor set assignment changed or changes its policy, the task/thread is assigned the limits imposed on that policy by the processor set (*processor_set_control* → **processor_set_policy_control**). Changing the limits for a processor set does not change the limits for currently assigned tasks/threads unless an option to the **processor_set_policy_control** call is used. The current limits for a policy for a processor set can be found with **processor_set_info**. Under no circumstances do the base attributes for a task/thread exceed its limits.

Each task/thread has an associated set of base scheduling attributes inherited from its parent (*task* → **task_policy** or *task* (*processor_set_control*) → **task_set_policy**). Changing the attributes for a task does not affect any contained threads unless an option to the set policy call is used. The privileged calls *task* (*processor_set_control*) → **task_set_policy** and *thread* (*processor_set_control*) → **thread_set_policy** allow a task/thread to set any base it wants (and the limit as well). A task/thread may select any attributes within its limits with *task* → **task_policy** or *thread* → **thread_policy**. When a task's/thread's processor set assignment is changed, it maintains its base attributes unless the changing of the task's/thread's limits as a result of the assignment requires adjusting the base attributes to the limits; if the task's/thread's policy is changed (to the processor set default) by the re-assignment, the base attributes are set to the processor set's default policy base set (established by *processor_set_control* → **processor_set_policy_control**). The current attributes for a task can be found with **task_info**, for a thread with **thread_info**.

Statistics concerning scheduling for a processor set can be found with *processor_set_control* → **processor_set_statistics**.

Clocks

Few of the kernel interfaces are sensitive to the passage of time. The basic IPC service does provide a time-out facility, but the typical use of IPC is to specify an infinite time or a zero time (zero to protect against a client whose reply port is full). The **thread_switch** interface also allows a time-out specification for scheduling depression. In either case, these interfaces provide indirect and imprecise access to kernel time services.

The kernel *clock* facility provides an explicit and well-defined interface to timeout services. The principal features of the clock facility are:

- Efficiency—The clock facility provides a basic, direct and efficient set of synchronous and asynchronous real-time alarm services.
- Granularity—The clock facility provides for dynamically changing the alarm resolution of the system. For example, the normal alarm resolution of the Mach kernel is 10-milliseconds since this is the minimum interval at which the kernel requires clock ser-

vice. If a given real-time application, however, requires alarms delivered to 1-millisecond accuracy, the clock facility can be utilized to meet that resolution.

- Relative/Absolute Alarms—The clock facility supports alarm services based on both relative (time interval measured from "now") and absolute time values.
- Timestamps—The clock facility provides timestamps on all alarms so that client applications can determine if requested time delays were achieved or not.

The clock facility implements (one or more) clock objects. Each host (or node) provides at least one clock, the REALTIME clock, which measures with (at least) moderate resolution the time since system initialization. This clock must provide alarm services. There may be other clocks, such as the standard defined BATTERY clock which has lower resolution and whose time measure is not affected by system power failures or service outages, and the standard defined HIGHRES clock which, if present, allows for high-resolution alarm services. The number and nature of the clocks exported by the kernel is typically a function of the clock/timer devices provided by the hardware, but any given implementation may provide other clocks abstracted from the hardware clocks.

A clock provides a monotonically increasing time value expressed in nanoseconds. The precision with which a clock measures the passage of time is expressed by a set of resolution values. A clock's *minimum* resolution expresses the finest precision that the clock can achieve in noting the passage of time. A clock's *maximum* resolution represents the coarsest precision at which the clock can be operated. A clock's *current* resolution, which must be a multiple of the clock's minimum resolution and less than or equal to the clock's maximum resolution, expresses the precision with which alarms and sleeps will be serviced by the clock.

The REALTIME clock provides a time reference whose initial value is set to zero by the kernel at system boot time. It is the responsibility of user-level code to translate the time tracked by the REALTIME clock into some notion of universal time, or to provide a user view of externally synchronized time. The BATTERY clock, in contrast, provides a time value that is controlled exclusively by user-level code and that survives power or system outages.

Each clock provided by the clock facility is named by both a name port (obtained by *host_name* → **host_get_clock_service**), which supports general clock functions, and a control port (obtained by *host_control* → **host_get_clock_control**), which is used only to set the time or resolution of the clock. A clock supports the following operations:

- *clock_name* → **clock_get_time** — Return the clock's current time.
- *clock_name* → **clock_get_attributes** — Return attributes of the clock's implementation or operation, such as the current resolution.
- *clock_name* → **clock_map_time** — Return a memory object (representative) naming a read-only memory object whose contents are the clock's running time value in a standard format. Not all clocks provide this service, but the REALTIME clock must.
- (system trap) **clock_sleep** (*clock_name*) — Delay the invoking thread until a given time. Not all clocks provide this service, but the REALTIME clock must.
- *clock_name* (*ledger*) → **clock_alarm** — Request a message to be sent at a given time. Not all clocks provide this service but the REALTIME clock must.

- *clock_control* → **clock_set_time** — Set the clock's current time.
- *clock_control* → **clock_set_attributes** — Set attributes of the clock's operation, such as the current resolution.

Hosts

Each uniprocessor or multiprocessor 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 clocks.

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. The result of **mach_host_self** is actually a "special" port defined for the task, but the typical (default) value is the name port for the host.

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 under System Initialization.

FIGURE 12 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.
- *host_control* → **host_get_boot_info** — Return operator supplied boot-time information.
- *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_processor_sets** — Return control and name ports to the defined processor sets.
- *host_name* → **host_processor_slots** — Return the physical slot numbers for the host's configured processors.
- *host_control* → **host_processors** — Return control ports for the host's processors.

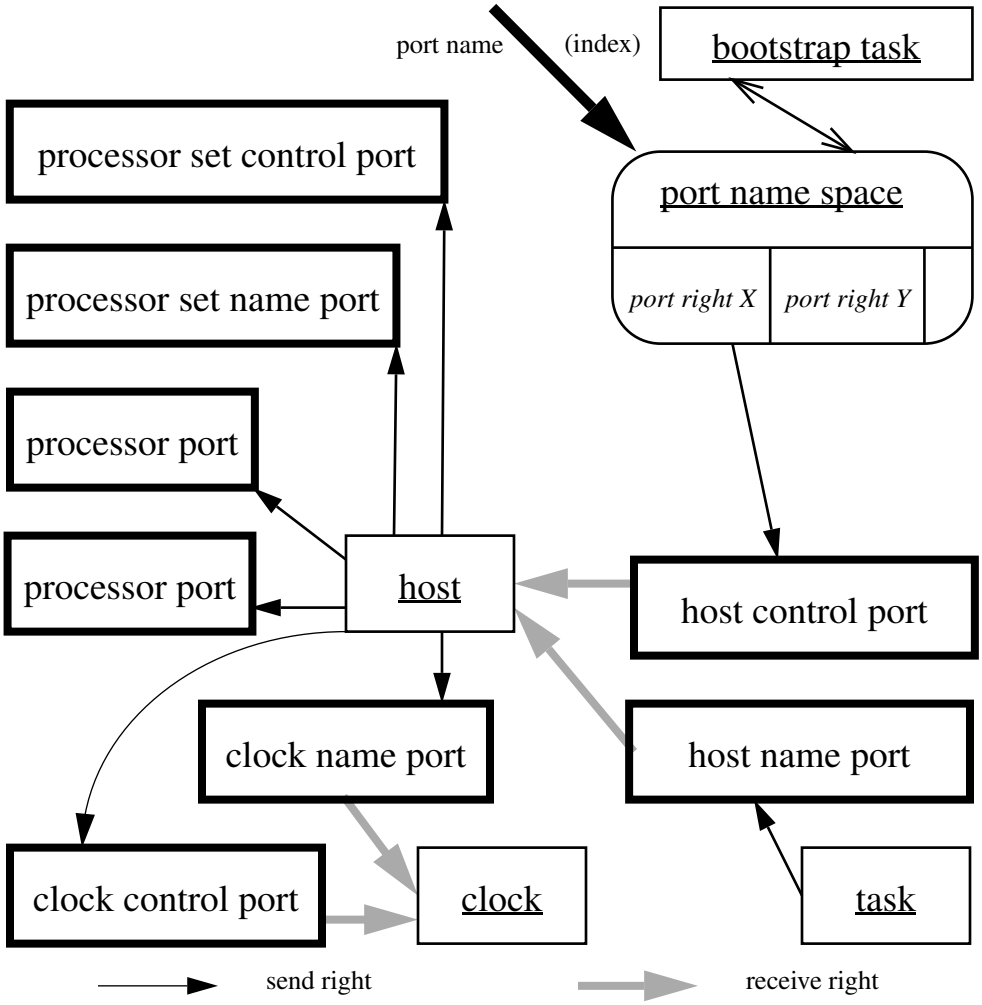

FIGURE 12 **Host (and Clock) Structures**

- *host_control* → **processor_set_default** — Return control and name ports for the host's default processor set.
- *host_name* → **host_get_clock_service** — Return the name port for a system clock.
- *host_control* → **host_get_clock_control** — Return the control port for a system clock.
- *host_control* → **host_statistics** — Return scheduling or virtual memory statistics for the host.

Nodes

The experimental "NORMA" support allows a collection of Mach hosts to be connected together to form a *multicomputer* system. In general, each host (called a *node*) in the multicomputer retains its own identity (its own set of processors, processor sets, devices,

name and control ports, etc.). However, certain mechanisms are transparently extended across the nodes:

- IPC—Messages between entities in the multicomputer transparently traverse the multicomputer. Notifications correctly reflect the distributed state of a port and its associated rights.
- VM—Shared memory is coherently maintained. A memory manager sees only a single logical kernel.

Since each node maintains its identity, to manipulate the resources of a node it is necessary to have access to the appropriate port for that node. Given the host control port for any node, it is possible to obtain (or set) the ports for the other nodes. These node specific ports are:

- A host control port, useful for deriving processor set and other related ports.
- A corresponding host name port.
- A device master port, used to obtain node specific devices.
- The default memory manager port.
- The root ledger ports.
- The host security port.

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

Currently, a task is created on a node and will not migrate from that node. A task is created on the same node as the task used as the parent in the **task_create** call. A task may be created explicitly on a given node with *task* (*ledger*) → **norma_task_create**. A similar function provided to simplify task migration, *task* (*ledger*) → **norma_task_clone**, has the same result except that the child always shares its memory regions with the parent (regardless of the parent's inheritance values); the child's inheritance values are set to be that of the parent

port_name_space [*port_name*] → **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.

Kernel 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* (*ledger*)→ **device_open**. (Normally, only some device mastering task would hold the device master port.) Each call to **device_open** returns a distinct port (optionally restricted to use by a single identity), the allowed operations on which being governed by the opening mode. 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 under System Initialization.

FIGURE 13 shows the client visible device structures. Access to the device master port provides access to all devices.

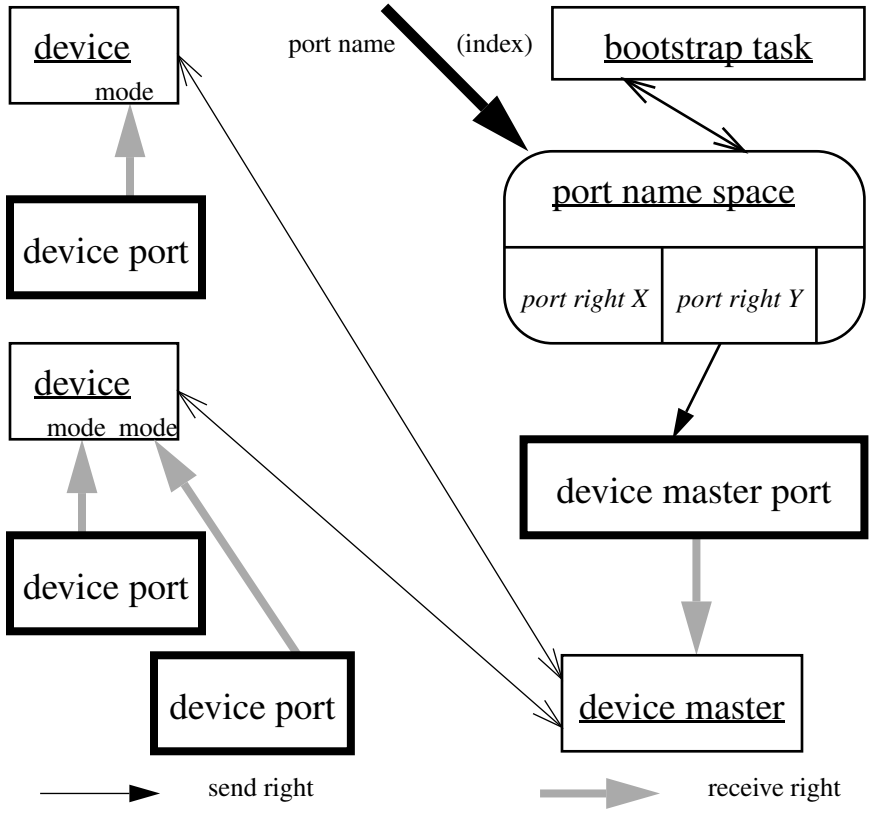

FIGURE 13 **Kernel Device Structures**

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 "inline" (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**. There is also a **device_read_overwrite** system trap (that is, it only applies in the case of a local device) which can read data directly into the caller's address space.

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 generates a new memory object (a memory object representative) for which the external memory manager is the device driver. Mapping this object (**vm_map**) provides an illusion of the device. The typical use of this call is when mapping a workstation's frame buffer into the display server's memory. Calling **device_close** will destroy such mapped windows. The allowed access modes to the mapped window and restricted use identity, if any, are governed by the associated device port attributes.

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).

CHAPTER 7

System Initialization

Once initialized, a Mach host continues to run until it fails (power loss, internal system panic, etc.) or until *host_control* → **host_reboot** is called. None of these situations involves an orderly shut-down of the tasks in the system; this is a responsibility of higher level software.

Since the Mach kernel itself maintains no permanent state, the subsequent (re-)initialization of a Mach host involves no recovery on the part of the microkernel. (System tasks, such as file servers, may well need to perform recovery of their permanent state.) The microkernel is simply loaded by the hardware bootstrap sequence and the microkernel initializes itself. The microkernel subsequently creates an initial bootstrap task, loads it and starts its execution. The kernel knows the existence of the image to load into this initial task and the starting offset of the image in some implementation defined way. The initial task is created with a single thread starting at the image's starting offset.

Bootstrap Task Properties

The bootstrap task is created with all of the privileges that the kernel exports, including access to the host control and device master ports as described below.

The initial state of the bootstrap task is either empty or not set, with the following exceptions:

- Virtual address space — contains the boot image, heap and stack.
- Scheduling parameters — timeshare policy, priority 96.
- Task self port — itself.
- Bootstrap port — kernel (described below).
- Host self port — local host name port.

- EXC_MACH_SYSCALL exception class exception port — host name port.
- Ledgers — the node root ledgers.
- Security token — SECURITY_KERNEL.

The job of the bootstrap task is to create and start the initial system servers. As the holder of all system privileges and port accesses, the bootstrap task passes to the servers those privileges and accesses that are necessary for their functioning.

Access to Privileged Ports

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 (*bootstrap* → **bootstrap_ports**). The reply from this request will contain the privileged port rights: the host control port, root ledger ports, host security port and the device master port.

(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.)

The holder of the host control port can obtain send rights to any port on the system (with the exception of device ports). The basic privilege mechanism provided by the kernel is the restriction of privileged operations to tasks holding control ports.

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 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).

APPENDIX A

Function Index

B

bootstrap_ports 80

C

catch_exception_raise 21, 22
catch_exception_raise_state 22
catch_exception_raise_state_identity 22
clock_alarm 66, 72
clock_get_attributes 72
clock_get_time 72
clock_map_time 72
clock_set_attributes 73
clock_set_time 73
clock_sleep 21, 72

D

default_pager_info 62
default_pager_object_create 62
default_pager_paging_file 62
device_close 76, 77
device_get_status 77
device_map 63, 77
device_open 76
device_read 77
device_read_inband 77
device_read_overwrite 77
device_set_filter 77

device_set_status. 77
device_write 77
device_write_inband. 77

H

host_get_boot_info 73
host_get_clock_control. 72, 74
host_get_clock_service. 72, 74
host_info 73
host_kernel_version 73
host_page_size 44
host_processor_set_priv 69
host_processor_sets. 69, 73
host_processor_slots. 67, 73
host_processors. 67, 73
host_reboot 73, 79
host_statistics 74

L

ledger_create. 67
ledger_get_remote. 66, 67
ledger_read 67
ledger_set_remote. 66
ledger_terminate 67
ledger_transfer 67

M

mach_host_self 21, 49, 73
mach_msg 21, 24, 31, 32, 35, 36, 37, 38
mach_msg_overtrap_trap 21
mach_port_allocate. 30, 33, 37
mach_port_allocate_name 30, 33, 37
mach_port_deallocate. 34, 35, 37
mach_port_destroy 34, 35, 37
mach_port_extract_right. 32, 36
mach_port_get_attributes 30, 34, 36
mach_port_get_refs. 35
mach_port_get_set_status. 37
mach_port_insert_right. 33
mach_port_mod_refs 34, 35
mach_port_move_member 37
mach_port_names. 36
mach_port_rename 36
mach_port_request_notification 31, 34, 35, 38
mach_port_set_attributes 30, 32
mach_port_set_mscount 36
mach_port_set_seqno 36
mach_port_type. 36
mach_ports_lookup. 26
mach_ports_register 26

mach_reply_port 21, 30, 33
mach_task_self 21
mach_thread_self 21
memory_object_change_attributes 49, 53
memory_object_change_completed 53
memory_object_create 62
memory_object_data_error 51, 54, 56
memory_object_data_initialize 62
memory_object_data_request 51, 55
memory_object_data_return 40, 51, 54, 56, 57, 62
memory_object_data_return_completed 40, 52, 55
memory_object_data_supply 51, 53, 54, 56
memory_object_data_unavailable 51, 56, 62
memory_object_data_unlock 54, 56, 57
memory_object_destroy 52
memory_object_establish 49, 50, 53, 60
memory_object_establish_priv 52
memory_object_get_attributes 49
memory_object_get_name 50
memory_object_lock_completed 54, 56, 57, 61
memory_object_lock_request 54, 56, 57, 61
memory_object_notify 49, 50, 52, 53
memory_object_rejected 49
memory_object_representative 50
memory_object_supply_completed 54
memory_object_synchronize 53
memory_object_synchronize_completed 53
memory_object_terminate 50, 52, 53

N

norma_get_special_port 75
norma_port_location_hint 75
norma_set_special_port 62, 75
norma_task_clone 75
norma_task_create 75

P

processor_assign 68, 69
processor_control 68
processor_exit 68
processor_get_assignment 68, 69
processor_info 68
processor_set_create 69
processor_set_default 69, 74
processor_set_destroy 69
processor_set_info 69, 70, 71
processor_set_policy_control 70, 71
processor_set_statistics 71
processor_set_tasks 69, 70

processor_set_threads 69, 70
processor_start 68

T

task_assign 27, 69
task_assign_default 27, 69
task_create. 26, 27, 44, 66, 75
task_create_security_token 27
task_get_assignment 27, 69
task_get_emulation_vector 23, 27
task_get_exception_ports 22
task_get_special_port 26
task_info 26, 70, 71
task_policy 27, 70, 71
task_resume. 27
task_sample. 27
task_set_emulation_vector 23, 27
task_set_exception_ports 22
task_set_policy 27, 70, 71
task_set_security_token 27
task_set_special_port 21, 26
task_suspend 27
task_swap_exception_ports 22
task_terminate 26
task_threads. 23, 27
thread_abort 21, 24, 37, 38
thread_abort_safely. 24
thread_assign. 24, 69
thread_assign_default 24, 69
thread_create 23, 27
thread_create_running. 23
thread_depress_abort 21
thread_get_assignment 24, 69
thread_get_exception_ports 22
thread_get_special_port 24
thread_get_state 23
thread_info 24, 70, 71
thread_policy. 24, 70, 71
thread_resume 23, 24
thread_sample 24
thread_set_exception_ports. 22
thread_set_policy 24, 70, 71
thread_set_special_port. 21, 24
thread_set_state. 22, 23, 24
thread_suspend 23, 24
thread_swap_exception_ports. 22
thread_switch 20, 21, 71
thread_terminate 23
thread_wire 24

V

vm_allocate 40, 45, 59, 62
vm_behavior_set 45
vm_copy 45
vm_deallocate 45, 46, 51
vm_inherit 44
vm_machine_attribute. 45
vm_map. 44, 45, 46, 47, 49, 50, 77
vm_msync 46, 53
vm_protect 45
vm_read. 45, 58
vm_region 44, 47, 50
vm_reserve 46
vm_set_default_memory_manager. 62, 73
vm_unreserve 46
vm_wire 45, 65
vm_write 40, 45, 58