MK++ Kernel
Executive Summary

OSF

November 17, 1995

Revision History:

Revision 1	November 19, 1993	
Revision 1.2	September 15, 1994	Consistency with Kernel Interface
Revision 1.3	May 5, 1995	Consistency with first release
Revision 1.4	November 17, 1995	Formal evaluation version

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

Contents

Tables and Figures

CHAPTER 1 Introduction

This book documents the user visible architecture of the MK++ kernel.

The MK++ kernel 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 MK++ include:

- Exploiting parallelism in both operating system and user applications.
- Supporting large, potentially sparse address spaces with flexible memory sharing.
- Compatibility with existing software environments (BSD, for example).
- Portability.
- Supporting advanced operating system requirements, such as trust and real-time.

MK++ as a foundation technology for diverse operating environments subscribes to the philosophy of:

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

It is a goal of MK++ that most traditional operating system functionality reside outside of the kernel, with almost everything being done by user mode tasks communicating via the kernel. Of course, the kernel must provide support besides task to task communication, in particular:

- Management of points of control (*threads*).

- Resource assignment (*tasks* and *ledgers*).
- Support of address spaces for tasks.
- Management of the physical hardware (memory, processors, device channels, clocks).

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

Kernel Abstractions

Each of the abstractions provided by the MK++ kernel 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:

- Task — The unit of resource allocation, providing a large address space and a port name (right) space.
- Thread — The unit of CPU utilization, lightweight (low overhead).
- Port — A communication channel. Ports are accessible only via send/receive capabilities, called port rights.
- Message — A collection of data objects transmitted via a port. These messages are tagged by the kernel with an unforgeable identity of the sender for use by servers in making access control decisions.
- Memory object — The abstract representation of virtual memory, the means by which tasks take control over memory management. The kernel provides the means by which control may be exercised over the identities that may access a memory object and the access modes by which they may access.
- Device — Virtual access to a kernel supported device. The kernel provides the means by which control may be exercised over the identities that may access a device and the access modes by which they may access.

Tasks and Threads

The MK++ 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 MK++ 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. MK++ wishes to support multiple points of control in a way separate from any given operating system environment's notion of process.

Instead, MK++ provides two notions: the *task* and the *thread*. A thread is MK++'s notion of the point of control. A task exists to provide resources for its contained 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 and port name 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. Portions of this space may be shared through inheritance or external memory management.
- Provides a port name space that contains the port rights granting access to system resources.
- 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 because a thread has little state (mostly its register state); its owning task bears the burden of resource management. On a multiprocessor it is possible for multiple threads in a task to execute in parallel. Even when parallelism is not the goal, multiple threads have the advantage 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 MK++ kernel provides the mechanisms to support large, potentially sparse virtual address spaces. Each task has an associated address map (maintained by the kernel) that 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 are 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 MK++ kernel does not implement all of the elements of this caching itself; it endeavors to enable user mode tasks to participate in these mechanisms.

A task’s virtual memory is not referenced via ports. Memory is referenced 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 MK++, many memory copy operations are actually *virtual* copy operations. Virtual copy operations are 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 MK++ kernel.

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 memory manager is responsible for determining the set of access modes to be permitted for various clients of the memory object; the kernel simply enforces these restrictions. The MK++ 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 MK++ philosophy. MK++ 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 MK++ kernel provides very few services of its own (for example, it provides no file service), a MK++ task needs to communicate with a potentially great many other tasks that do provide these services. These communication channels in MK++ are called *ports*. A port is a unidirectional channel consisting of a queue that holds *messages*. A message is a collection of data, memory regions and port rights. A port *right* is a capability 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 can acquire a port right only via one of two ways:

- Some other task holding the right, or the kernel, sends the right in a message.
- Some other task holding a right to the task port for the task explicitly inserts a right in or extracts a right from that task.

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. This message is tagged by the kernel with an unforgeable identity of the sending task. At some later time, the task with receive rights to that port performs a message-receive operation. Note that this message transfer is an asynchronous operation. The message is logically copied into the receiving task. Multiple threads within the receiving task can be attempting to receive messages from a given port, but only one thread receives any given message.

Architectural Elements

The MK++ kernel provides an environment consisting of the following elements:

- thread — An execution point. A thread is a light-weight entity; most of the state pertinent to a thread is associated with its containing task.
- task — A container to hold references to resources in the form of a port name space, a virtual address space and a set of threads.
- port — A unidirectional communication channel between tasks, accessible only via port rights.
- port set — A set of ports that 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 entries, each of which names a port right or port set.
- message — A collection of data, memory regions and port rights passed between two tasks, along with kernel applied data, such as the sender's security ID.
- 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. This object is implemented by a user task called a memory manager that holds receive rights to the abstract memory object port, 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 names the client that is permitted access as well as the potential access modes permitted to that client. The manager for the memory object holds receive rights to these ports.
- 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 client's visible memory image.
- processor set — A (the) set of processors, each of which can be used to execute threads.
- ledger — A resource accounting device that counts and limits the amount of kernel resources that may be consumed.
- host — The multiprocessor as a whole.

- clock — A representation of the passage of time (a time value incremented at a constant frequency).
- alarm — A kernel object that sends a message at regular time intervals.
- device — A kernel object representing a physical device that is accessible by a user mode task.

Each of these notions will be discussed in detail in this book.

Kernel and Object Invocation

To reference any resource other than the set of hardware registers and the virtual address space, a thread must execute a special trap instruction that causes the kernel to perform operations on behalf of the thread, or causes the kernel to transmit a message to some agent that acts 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 implement resources (those listed earlier) 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 and port name space, all system resources are accessed through a level of indirection known as a *port*. A port is a unidirectional communication channel between a client that requests a service and a server that 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, in order 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 port and specifies the rights permitted for that port. A task may hold multiple rights that name the same 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.

Document Conventions

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 is 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 creating a task given a specified ledger and "prototype" task); 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 the 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**

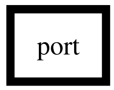
is used to show that an *id* is required as a parameter in the message to indicate which range or element of *object* is to be manipulated.

The parenthesized notation,

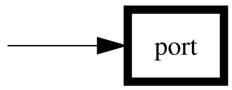
object (*port*) → **function**

is used to show that the (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.

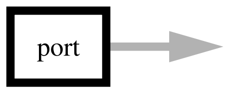
Figure 1 shows the conventions used in diagrams showing kernel structures.



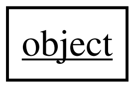
Port. This representation is used for kernel ports (those that name kernel objects) as well as user ports.



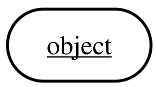
Indicates something (on the opposite side of the arrow) that holds a send right to a port.



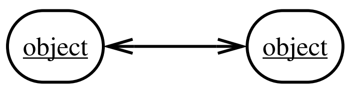
Indicates something (at the arrow) that holds the receive right to a port.



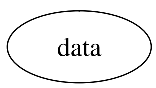
A first class kernel object — one named by kernel ports and directly manipulable by client tasks.



A non-first class kernel object — an object manipulable by client tasks, but only by manipulating the first class kernel object to which it is bound.



Indicates that two objects are bound or refer to each other in a way other than by possessing port rights to each other.



A random kernel data structure, such as a message.

Figure 1 **Structure Diagram Conventions**

CHAPTER 2 Inter-Component Transactions

The MK++ kernel constitutes the lowest layer in the system structure. As such, it has no dependencies on any user mode task and never waits for a user task. (User threads, though, do often wait via kernel mechanisms.)

All invocations of the kernel consist of some thread generating some hardware exception. The exception may be a true exception (invalid virtual address), a page fault, or a kernel service trap. Each service trap has an associated trap number by which the kernel can determine what operation is being requested.

One of these service traps is the **mach_msg** trap, by which most kernel invocations are made (as well as service invocations of other tasks). If the **mach_msg** trap is called with the MACH_SEND_MSG option and the destination right names a kernel port, the trap is a kernel service invocation (called a kernel *function* in the documentation).

All kernel trap invocations are inherently synchronous with respect to the invoking thread; the thread waits until the kernel takes some action. In the case of kernel service invocations, the **mach_msg** trap does not return until the kernel has performed the desired operation and returned the results (in the form of a message).

The kernel does participate in a variety of asynchronous message interactions with user mode tasks. In these exchanges, the user mode task sends a message to the kernel; the kernel returns no message or result at that time. Such invocations typically cause the kernel to take some (possibly delayed) action and the kernel typically sends an explicit response or some other messages at a later time (such messages are called *service interfaces* in the documentation.) All messages from the kernel are tagged with the kernel's identity.

The following is a classification of the message interactions.

- Trusted computing base (TCB) or non-TCB client invoking a non-privileged kernel service, with the kernel returning a reply to the client.
- TCB client invoking a privileged kernel service, with the kernel returning a reply to the client. The privileged or protected services are divided into the following sets:
    - Bootstrap server using the system bootstrap port to obtain the other privileged ports.
    - Device server using the device master port.
    - Host control server using the host control, (default) processor set control, or clock control ports.
    - Subject server using the ledger control or host security ports.
- TCB or non-TCB client invoking an external memory management function.
- Kernel sending an asynchronous external memory management service interface message to some TCB or non-TCB client.
- Kernel sending an asynchronous default memory management service interface message to the default memory manager over the abstract backing space port. The interactions between the default memory manager and the kernel are the same as for any memory manager but take place over the virtual paging object port.
- Kernel sending an alarm message to some TCB or non-TCB client.
- Kernel sending an exception message to some TCB or non-TCB client.
- Kernel sending an IPC notification message to some TCB or non-TCB client.
- Kernel sending a network device filter packet message to some TCB or non-TCB client.
- Kernel sending a program counter (PC) sample message to some TCB or non-TCB client.

Of the interfaces described in *Kernel Interfaces*, all of them are TCB interfaces with the exception of the privileged and protected interfaces.

CHAPTER 3 General MK++ Manipulations

The MK++ kernel objects and their manipulations are divided in this book into three groups:

- General (unprivileged) manipulations, described in this chapter.
- The external memory management protocols, described in CHAPTER 4.
- Privileged and control manipulations, discussed in CHAPTER 5.

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 a way governed 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 its memory space. An attempt to execute a privileged machine instruction causes an exception (discussed later) as will any other fault or illegal instruction.

To affect the structure of the address space, or to reference any resource other than the hardware registers or virtual 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.

Figure 2 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 (typically) or via the containing task.

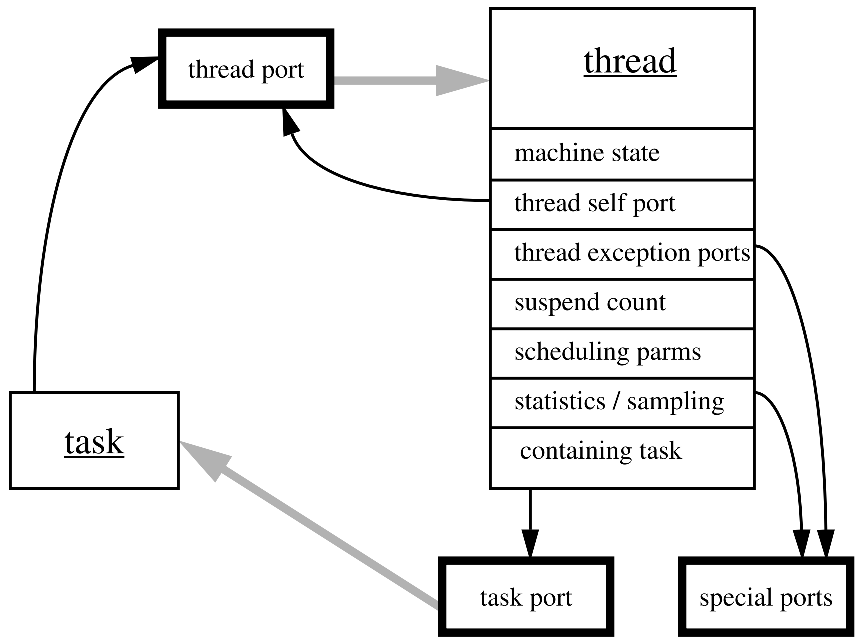


Figure 2 **Thread Structures**

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 MK++ 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 MK++ 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 that are used to select the threads that are to execute within its 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. One of the options to **thread_switch** causes the scheduling attributes of the thread to be adjusted so that other threads 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, or the trap **thread_depress_abort** is called.

Finally, the **clock_sleep** trap causes the thread to be delayed until a specified time.

Identity Traps

Other than the few traps mentioned in this chapter, all other requests for services require a port right. Even requests to the kernel that manipulate the current thread or task need a port right (naming the current thread or task). To initialize 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 that were 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 set for the task (the TASK_HOST_NAME_PORT task special port).

Reply Port Trap

The **mach_reply_port** trap is also needed for task initialization 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 calls.

Message Send and Receive Trap

The final and most important trap is **mach_msg_trap** (the trap invoked by the **mach_msg** library routine). This is the trap that provides access to all other system services. It sends and/or receives a message to/from a port. Messages may be sent to or received from only those ports for which the task holds appropriate (send or receive) rights. The semantics of this call are very involved, and described in detail in the *Kernel Interface*s document, as well as various sections of this document.

Exception Processing

When an exception occurs in a thread (these exceptions are listed under **catch_exception_raise** in the *Kernel Interfaces* document), the thread, executing in kernel context, sends a message, the contents of which describe the exception to an exception port.

For any given exception, there are two settable 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. A task inherits its parent's exception ports. A newly created thread's exception ports are set to NULL. The kernel selects the first of these ports (in the order listed; i.e., thread, then task) that is defined as the destination of the exception message. If neither port is defined, 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.
- *exception_port* → **catch_exception_raise_state_identity**—The **catch_exception_raise_state** message with the inclusion of the task and thread self ports.

When the exception handling task has handled the exception, the thread is restarted by calling *thread* → **thread_exception_continue** or *thread* → **thread_exception_continue_state**, depending on whether the thread's machine state should be modified as part of restarting it. The thread's state can also be explicitly set before restarting it with *thread* → **thread_set_state**.

Lastly, the thread can be restarted by invoking the trap **thread_continue_state**. This trap is an optimization that enables the caller to continue a thread with the specified state and optionally, receive a message (e.g., wait for another exception).

Not every exceptional condition that a thread encounters is handled as described above.

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 in the next chapter).

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 MK++ 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.

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 user mode 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. Three steps are needed to obtain a deterministic result.

- *thread* → **thread_suspend** is used to stop the thread. This step is unnecessary if the thread has just been created and has yet to run. It is needed, though, for exception processing or for asynchronous interruption (such as signal delivery). The result of the suspend call is to cause the thread to make no more progress executing user mode instructions (it does, however, complete any kernel invocation, or wait for a message reply or page while suspended).
- *thread* → **thread_set_state** can then be safely used.
- *thread* → **thread_resume** restarts the thread.

The return from the **thread_suspend** call guarantees that the target thread can make no further progress in user space. It does not guarantee that the thread will not continue to execute in kernel space to complete any operation in progress. This may include the act of taking an exception. If it is necessary to synchronize with such activities, *thread* → **thread_notify** is used. This call causes the thread to raise a special "notify" exception the next time the thread is about to leave kernel mode. (If the thread is suspended, it raises the exception but remains suspended.)

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**.

thread → **thread_sample** provides support for the Posix **profil** system call. After being called, the kernel starts sampling the program counter (PC) of the specified thread periodically (whenever it is running when a clock interrupt occurs). Buffers of these PC values are sent, when full, to a port specified in the sample call.

The scheduling attributes of a thread can also be set and examined as described under processor sets.

Tasks

A task can be viewed as a container that holds a set of threads. Most important, it contains those elements that its contained threads need to execute: a port name space and a virtual address space.

Figure 3 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 or the contained threads.

Life and Death

A new task is created with *ledger_name*, *task* → **task_create**. Note that task creation is an operation referencing an existing (parent or “prototype”) task. The new task has the same security ID as the prototype task.

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 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:

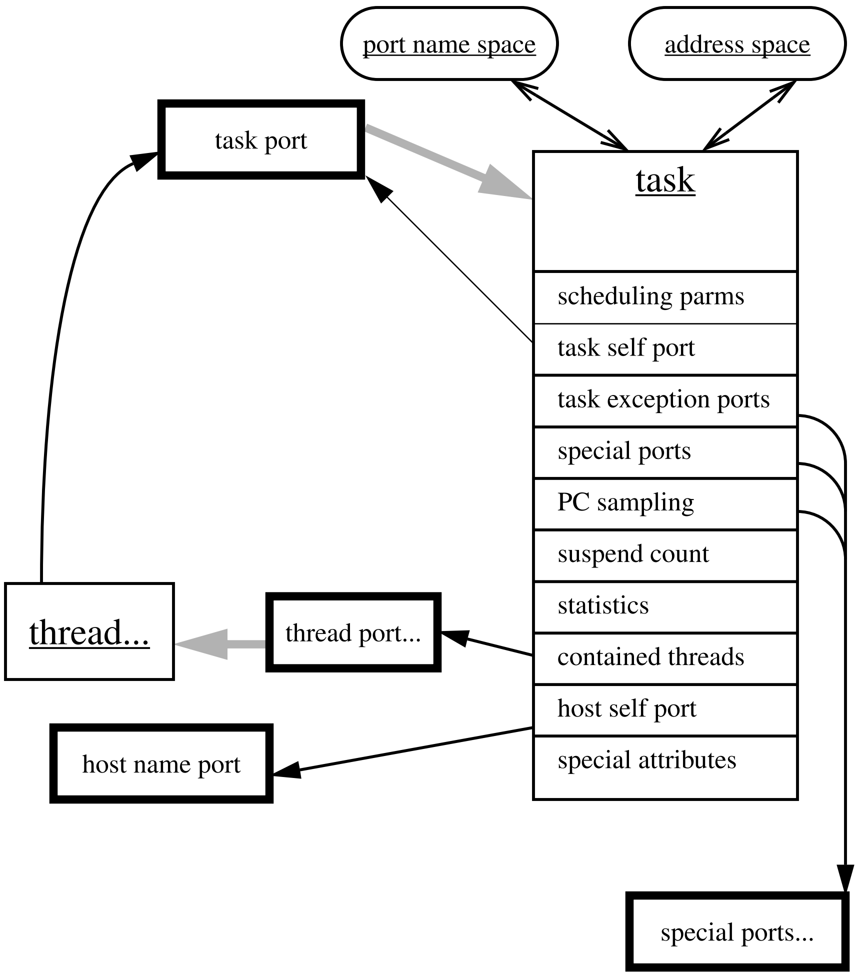


Figure 3 **Task Structures**

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

Thread Management

A thread belongs to one and only one task. A task does not itself execute; however, for convenience, some execution properties can be set for a task that 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 manipulate the task's suspend count as well as the contained threads' suspend counts. Newly created threads inherit the parent task's suspend count. However, a thread can execute only when its suspend count is zero, irrespective of the suspend count of its containing task.

The default scheduling properties for threads can be set and examined as discussed under processor sets.

task → **task_sample** provides support for the Posix **profil** system call. After being called, the kernel starts sampling the program counter (PC) of all threads within the task periodically (whenever one is running when a clock interrupt occurs). When full, buffers of these PC values are sent 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 an unforgeable security ID. The kernel tags all messages sent by the task with this identity for use by TCB servers in access control decisions. This identity is also checked by the kernel when the task attempts to access memory objects or devices. The security ID is inherited from the parent task by **task_create**, or explicitly set via privileged task creation (**task_create_security_id**).

Ports

A port is a unidirectional communication channel between a client that requests a service and a server that 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
- Port right and out-of-line memory size limits.

Figure 4 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

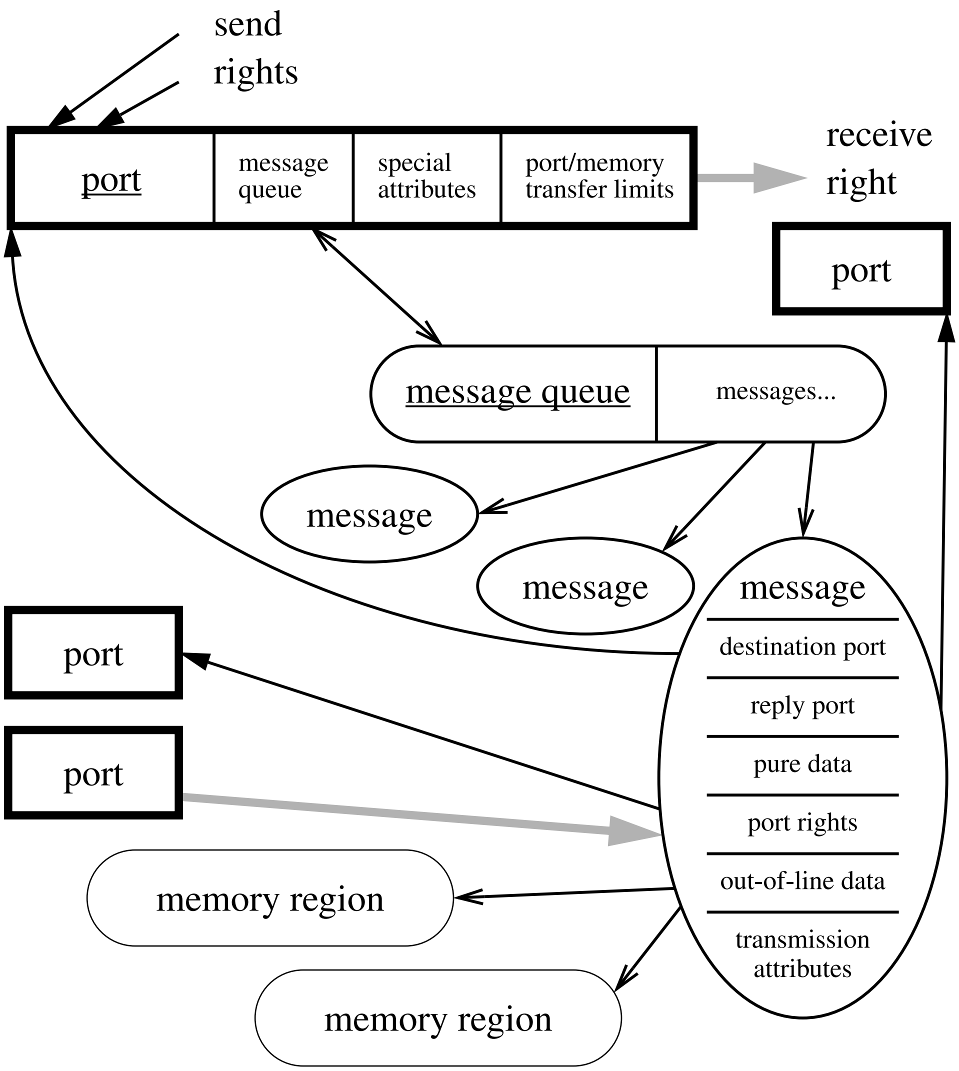


Figure 4 **Port Structures**

messages is shown in detail, with its destination port, reply port reference, a send and a receive right being passed in the message, as well as some out-of-line memory.

Few operations affect the port itself. Most operations affect port rights or a port name space containing those rights, or they may affect the message queue. Even then, the port or its contained messages can only be manipulated by a task holding appropriate port rights.

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**. Such a port has its kernel resources drawn from the target task's ledger. A port can be created drawn against a specific ledger with *ledger_name*, *port_name_space* → **ledger_create_port**.

A port is destroyed only when the receive right is destroyed, or its owning ledger 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. A request for such a notification is made 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" (is now unusable) because of the destruction of the named port. The message indicates the task's name for the now-dead port.

Messages

A message is a collection of data passed between two entities. This collection may include untyped data bits, (out-of-line) memory regions, and port rights. 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 it is received.

Besides pure data, a message can contain port rights. This is very significant. It is by receiving them in a message that a task obtains new rights (with the exception of the **mach_port_insert_right** and **mach_port_extract_right** calls against a task port).

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. One of these attributes is the sender's security ID. (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 and port rights) and some additional message body data. The message body data normally contains type conversion information as well as random data 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 or making a copy of it, or generating a right from a right of a different type.

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.

Neither the values in the out-of-line data regions nor those in the message data are typed by the kernel. 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.

Message Queues

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

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 a kernel). Messages sent to a port via send-once rights are unordered. (Port rights are explained in the section, **Port Rights**.)

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. This type of port right is an optimization (no deallocation required).

Port rights are a secure, location-independent way of identifying ports. The rights are kernel-protected entities; clients manipulate port rights only by the 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.

Option flags in a message will cause **mach_msg** to make a copy of an existing send right, or to generate a send or a send-once right from a receive right. Rights can also be forcefully copied or moved by *port_name_space* [*port_name*] → **mach_port_extract_right** (the equivalent of the target sending the right in a message) and *port_name_space* → **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 5 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.

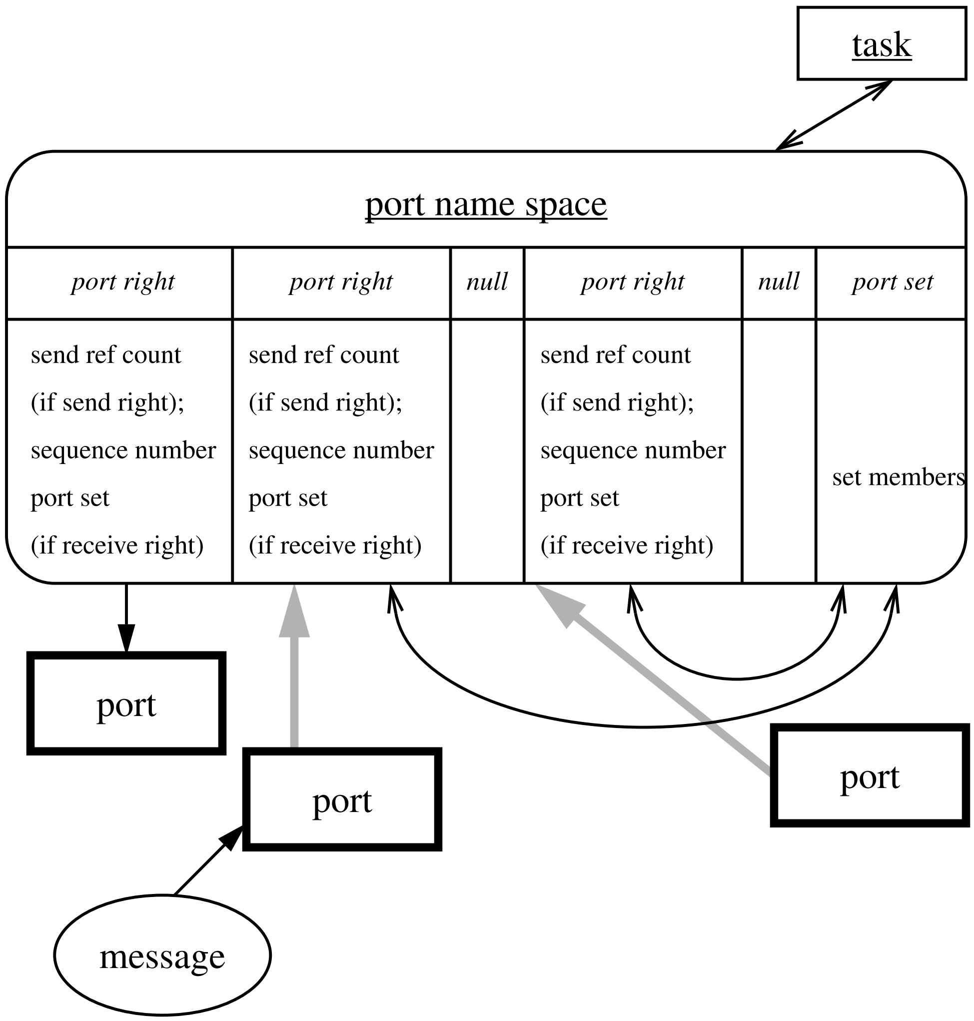


Figure 5 **Port Right Structures**

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* → **mach_port_allocate**. A port right is destroyed explicitly by *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**.

Send-Once Rights

A send-once right allows a single message to be sent. These rights are generated only from the receive right. A send-once right guarantees that a message results 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 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.

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 is associated with a port name space.

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

- MACH_PORT_NULL — No associated port right.
- A port right — A send-once, send, or receive right for a port.
- A port set name — A name that 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 distinct name for each distinct right. Receive and send rights 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 notification requested, discussed below) associated with the name. If the port name space also holds the receive right, that receive right has the same name as the send right.

Whenever a task acquires a right (by whatever means) it is assigned a port name, subject to the rules described above. 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. Calling *port_name_space* [*port_name*] → **mach_port_mod_refs** modifies the reference count for a specified right type associated with a 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.

A name becomes unusable when its associated port is destroyed. A task can be notified (a message sent to it) when one of its names becomes unusable ("dead") via *port_name_space* [*port_name*] → **mach_port_request_notification**. Requesting a dead name notification sets a "dead name" reference count, which is cleared only by cancelling the request or by explicit modification of the reference count if the dead name notification had been generated. This additional "reference" to the name avoids a race with other threads manipulating the name.

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

- Naming a right to a live port
- Naming a right to a dead (non-existent) port, with a dead-name notification having been sent when the name became dead
- Not naming a right, 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**.

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

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.

When a message is de-queued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The de-queue and increment operations are atomic, so that multiple threads receiving messages from a port can use the *msgh_seqno* field to reconstruct the original order of the messages.

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

Port Sets

A port set is a set of ports that 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 is received at random from one of the ports in the set. Each of the receive rights in the set has its own name and the set has its own name.

A receive operation performed 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* → **mach_port_allocate**. It is destroyed by *port_name_space* [*port_name*] → **mach_port_destroy** or *port_name_space* [*port_name*] → **mach_port_mod_refs** (a port set is considered to have a "port set" reference count).

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

Message Transmission

The **mach_msg** trap sends and receives MK++ 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 message is tagged with the security ID of the sending task.

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 destroyed.

A received message can contain port rights and/or 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.

Each received out-of-line memory region is dynamically allocated as temporary memory in the receiver's address space (as if by **vm_allocate**). The received data appears in allocated space at the same alignment within a page as it appears in that of the sender. When data is received, the regions are pageable (by default), but can optionally be requested to be wired.

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 handled specially.

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

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 threads can be set into execution.

An option to the **task_create** call allows 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 internal 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 explicitly 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 affect directly 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. The memory object representative port is a port whose receive right is held by the memory manager that will implement the abstract memory object. By appropriate use of these representative ports (described in CHAPTER 4), the memory manager can restrict access to the memory object to particular client identities as well as limiting the access modes permitted to each identity. The kernel will permit the memory object to be mapped only if the representative port permits the client to map the object with the requested access modes.

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 boot-loads. Calling *virtual_address_space* [*memory_range*] → **vm_allocate** can be viewed as a simplified form of calling **vm_map** when the default memory manager is desired.

Specific 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 performance 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 restrict further the allowed permissions, as well as its current protection mask.

- *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 regions wiring the pages and does not un-wire them until the count returns to zero.
- *virtual_address_space* [*memory_range*] → **vm_unwire** — Remove the wire flag for the region.
- *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 (fetch) 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.

There is also a trap version for read and write, **vm_read_inline** and **vm_write_inline**, that can read and write small amounts with considerably less overhead than their out-of-line counterparts.

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**) and dynamically allocated out-of-line memory received in IPC messages appear at random addresses.

In some environments it is important to identify the areas in which such allocation must not occur, in order 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**. Calling *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 remains reserved when that object is de-allocated. Inheritance can be set for reserved regions and reserved regions can be inherited.

Kernel Devices

The device interface is inherently asynchronous — a client sends a message to a device; a reply message indicates status or data returned. There exist, though, MK++ Interface

Generator (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.

Devices are named by device ports, one per **device_open** operation per device. Privilege is required to create a device port; this is described in CHAPTER 5. The holder of a device port may terminate its use with *device* → **device_close**. Each device port has an optional security tag that restricts use of the device port to tasks with a matching security ID. Each device port also restricts operations to the access modes permitted by that device port.

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 that uses out-of-line virtual memory managed space and another that transmits data "in-line" (inband) in the message (efficient for small transfers). These calls are *device* → **device_read**, *device* → **device_read_inband**, *device* → **device_write** and *device* → **device_write_inband**.

Each device also defines some status information that can be read by *device* → **device_get_status** or 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 (but only one per device port) and each packet may well be selected by and 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 user access to the device registers and/or memory. The actual access mechanism (memory mapped, I/O instruction, etc.) is platform- and device-specific. The typical use of this call is to map a workstation's frame buffer into the display server's memory. Calling **device_close** destroys 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 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 MK++ IPC message (and incur the associated costs, especially the copy of data). Closing a device port implicitly revokes access to the mapped representatives obtained via that opening.

Clocks

The kernel *clock* facility provides an explicit and well-defined interface to time 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 MK++ kernel is 10-milliseconds since this is the minimum interval at which the kernel requires clock service. If a given real-time application, however, requires alarms delivered to 1-millisecond accuracy, the clock facility can be used 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.
- Time-stamps—The clock facility provides time-stamps on all alarms so that client applications can determine if requested time delays are being achieved or not.

The clock facility implements one or more clock objects. Each host 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 are 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 are to 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 a name port (obtained by *host_name* → **host_get_clock_service**) that supports general clock functions. 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.
- *ledger_name*, *clock_name* → **alarm_create** — Create an alarm object. Not all clocks provide this service but the REALTIME clock must.

There is also a trap, **get_time_stamp**, which returns the real-time clock value, with sufficient accuracy and timeliness to be usable as a time-stamp.

Alarms

An alarm object (created via **alarm_create**) is a source of alarm notification messages (**alarm_reply**) sent at specifiable times. When created, an alarm object is initially unarmed. The **alarm_set** call arms the alarm; at a specified future time (or periodically starting at some future time), alarm notification messages will be sent. If the alarm is already armed when **alarm_set** is called, the previous alarm is cancelled (with an appropriate alarm notification sent to this effect). An alarm can be explicitly disarmed by **alarm_cancel**. Destroying an alarm object (**alarm_terminate**) also cancels an outstanding alarm.

Processor Sets

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

In general, MK++ tasks are not concerned about their assigned processor set, unless they are concerned about their scheduling.

Currently there is only a single processor set (the default). Its name port can be obtained with *host_name* → **processor_set_default**.

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

Given only one processor set, all tasks and threads are assigned to it. (For a task, being assigned to a processor set means that, by default, any new threads in that task will be assigned to that processor set. For a thread, being assigned to a processor set means that the thread executes only on processors within that set.) 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.

Any given thread is associated with 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 that it inherits from its containing task. The task's policy can be viewed with **task_info**; the thread's, with **thread_info**. Changing the policy for a task does not affect any contained threads unless an option to the set policy call is used.

Since some of the implemented scheduling policies permit threads to over-consume processor time, a subset of the implemented policies are typically enabled for random tasks/threads. The currently enabled set can be found with *processor_set_name* → **processor_set_info**. A task may select a policy from this subset of enabled policies with *task* → **task_policy**; a thread, with *thread* → **thread_policy**.

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 changes its policy, the task/thread is assigned the limits imposed on that policy by the processor set. The current limits for a policy for a processor set can be found with *processor_set_name* → **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. Changing the attributes for a task does not affect any contained threads unless an option to the set policy call is used. A task/thread may select any attributes within its limits with *task* → **task_policy** or *thread* → **thread_policy**. The current attributes for a task can be found with *task* → **task_info**; for a thread, with *thread* → **thread_info**.

Hosts

A host has its own physical memory pool, set of devices, default memory manager, threads and tasks, and clocks. Whereas many of these objects can be manipulated by random tasks, the host as a whole is typically manipulated only by privileged tasks.

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.

The following host specific calls are provided:

- *host_name* → **host_info** — Return host information.
- *host_name* → **host_kernel_version** — Return the kernel version.
- *host_name* → **host_get_clock_service** — Return the name port for a system clock.

- *host_name* → **processor_set_default** — Return the name port for the host's default processor set.
- *host_name* → **memory_object_establish** — Used as part of external memory management initialization.
- *host_name* → **host_page_size** — Return the virtual memory page size.

CHAPTER 4 External Memory Management

The MK++ 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 that names the memory object to back that range, that names the task's identity, and that permits the task's desired access modes (*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 releases 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, since the kernel cannot be stalled by a memory manager and because memory managers wish the maximum possible concurrency in their operations.

The kernel's messages are sent to the memory manager via the memory object's abstract memory object port; messages to the kernel are sent via the memory object's (kernel) memory cache control port. (The messages sent by the kernel are sent via routines labeled as "Server Interfaces" in the *Kernel Interfaces* 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, as well as the client identity permitted to use that representative and the access modes allowed to the client. 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 that 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).

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.

Figure 6 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 is to send subsequent manipulation messages and the kernel must communicate to the memory manager the memory cache control port over which the manager is to send subsequent manipulation messages.

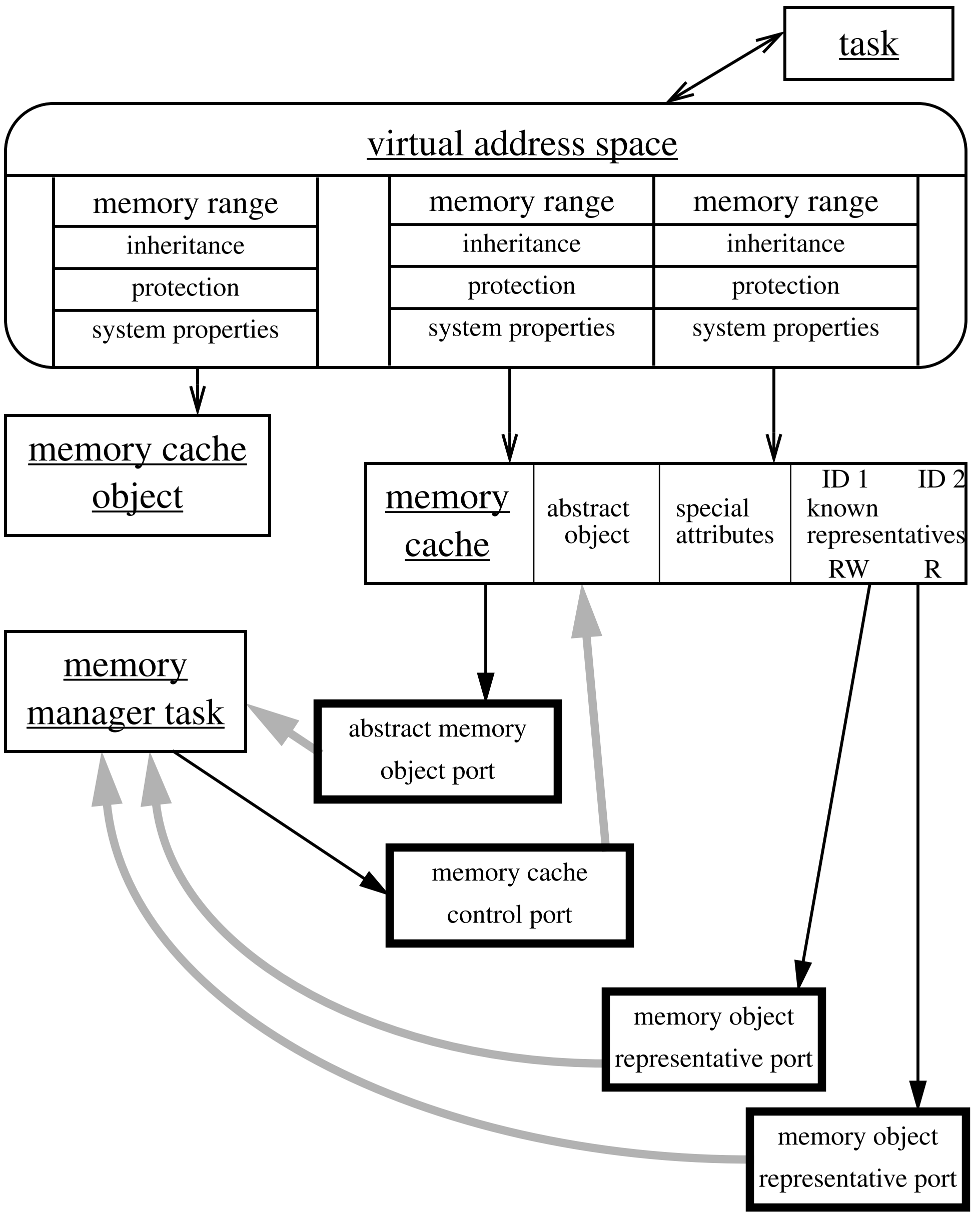


Figure 6 **Virtual Memory Structures**

Since the abstract memory object port is the port over which the kernel sends messages according to specially designed protocols, it is not a good idea for clients also to 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 that, in turn, imply the associated abstract memory object port, allowed client identity, and associated set of allowed access modes to the associated memory object.

Both abstract memory object ports and memory object representative ports are normal MK++ ports (that is, their receiver is a MK++ 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** and the kernel does not yet know the identity of the object so represented, the kernel engages 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.

This special dialog consists of the following steps:

1. Some task issues *task* → **vm_map** (*memory_object_representative*).
2. *task*'s kernel sends *memory_object_representative* (*host_name*) → **memory_object_notify** (*memory_cache_control*).
3. Assuming that the memory manager accepts that the supplied *host_name* names a host (either from **mach_host_self** or by querying a name server), the manager responds with *host_name*, *ledger_name* (*memory_cache_control*, *memory_object_representative*) → **memory_object_establish** (*abstract_memory_object*, *allowed_client_identity*, *allowed_access_modes*).
4. Assuming that the *memory_object_representative* port matches that originally named in the **vm_map** call and that the *memory_cache_control* port is the one created when that representative was mapped, the kernel accepts the initialization. In this case, the kernel sends no explicit reply, but eventually sends some message to the *abstract_memory_object* port, all of these messages including the *memory_cache_control* port as well. If the kernel does not accept the initialization, it sends *abstract_memory_object* → **memory_object_rejected** (*memory_cache_control*). This can happen either because incorrect ports were supplied in the establish call (an authentication failure) or the kernel terminated the initialization.

The purpose of this dialog is for the external memory manager and the mapping kernel to mutually authenticate each other — to establish protected communication between them. That is, only the kernel and the memory manager should hold rights to the abstract memory object and memory cache control ports.

In Step 1, a random task maps a random port.

In Step 2, the kernel creates a memory cache control port and gives it out. Since the kernel can't know that the port to which it sent the notify message was a memory object representative port, the kernel must at this point suspect that any and all tasks are given this memory cache control port. The kernel can never be sure that only the manager holds this right (the kernel can't stop the manager from giving it away).

However, at this point the kernel has an internal binding between the random port (the representative) supplied and the cache control port given out. Any task that can present both of these ports is deemed to be the memory manager by the kernel. If the task that holds receive rights to the port into which the kernel sends the notify message does not give away the cache control port, then that task is protected from interference when manipulating the cache object.

In Step 2, the manager is presented with what may be a memory cache control port. It may not be, though, since any and all tasks hold the memory object representative port and can send arbitrary messages. The supplied port may even be the cache control port for some other memory object.

What the manager can know is that the host name port given to it really is one, so it knows when it sends its establish message that only the kernel is to be given rights to the abstract memory object port.

At this point the manager knows that it is free from interference when the kernel manipulates it. The manager does not yet know, though, that it has the proper memory cache control port, nor whether any other task has it as well. However, since the manager gave away the abstract memory object port only to a kernel, the manager will accept any message coming over that port as a true statement from that kernel.

In Step 3, the kernel is presented with both the representative port and the cache control port, so the kernel deems the establish message as truly from the manager, accepting the supplied abstract memory object port as the true abstract port for the object.

By receiving a message other than **memory_object_rejected** in Step 4 (which the manager accepts as true), the manager knows that the kernel has accepted the manager as the manager for the object. Since the manager did not give away the cache control port (except to the kernel), and the kernel only sent it to the representative port that the manager holds, the manager now knows that only it holds the cache control right.

By the way, 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 **memory_object_establish** message carries object attributes that 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**.

This sequence involving the host name port and a memory cache control port / memory object representative port pair 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 with 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*) → **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 destroys 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, for this message to 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 discards the **memory_object_representative** message (since the object is not initialized from the kernel's point of view).

The manager detects 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 were to accept it and then immediately forget the association.

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 (by any task) 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. Alternatively, the memory manager can reply with *memory_cache_control* → **memory_object_data_unavailable**. In this case, the kernel supplies the missing data (zeroes). 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**.

Once pages are sent to a memory manager, the kernel is free to request them again at any time, possibly immediately. 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.

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.

It is possible to effectively force all tasks holding mappings through a representative to give up their access with *memory_cache_control* → **memory_object_revoke**. This action on the part of a memory manager does not actually unmap the object from all tasks, but it does cause any future accesses to the memory to fail.

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.

Alternatively, the memory manager may call *memory_cache_control* → **memory_object_destroy** to explicitly shut down the memory object. All resident pages are discarded and no more activity is allowed for the memory object. The kernel responds 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.

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 unmapped state, the memory object is 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 is sent, otherwise only a **memory_object_terminate** message is sent.

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 (unmodified) 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.

The 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 returns 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 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**. (Attempts to access memory with a mode not permitted by the mapping memory object representative do not do this; such accesses always cause memory access exceptions.) If the memory manager honors this request, it responds with **memory_object_lock_request** to specify the new allowed permissions.

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 for an **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 tables.

A page can be in the following states:

- *out*— not in memory, no task waiting for the page.
- *fetch* — not in memory, some task waiting for the page. A page enters this state when some thread "peeks" (attempts to read) or "pokes" (attempts to write) the page while it is in the *out* state. When such an access is attempted, the kernel sends a **memory_object_data_request** message to the manager requesting the page with whatever access modes are necessary to satisfy the thread's attempted access (assuming the task's mapping permits it). Once a page is in this state, all threads attempting to access the page stalls while waiting for it to be supplied. The kernel sends no additional messages to the manager. Once the page is supplied (the page goes to an in-memory state), the accessing threads reattempt their accesses, causing whatever kernel actions are implied.
- *in-memory* — A page in memory and accessible to tasks is labeled with a tuple: <*modified*, *current access*, *desired access*, *precious*>, where *modified* is true (not overlined) if the page has been modified since last requested from the manager or returned to the manager, *current access* indicates the access modes that will not fault, *desired access* indicates the access modes requested by threads attempting to access the page, and *precious* is true (not overlined) if marked as precious by the manager. Note that

current access is always less than or equal to *desired access* and that the *modified* flag can only be set via a “poke” operation when the page has write access.

Note that *wired* is not shown in the page states. Being wired keeps the page from being evicted. Since eviction is only kernel-initiated, the presence of the *wired* bit can be viewed as if the kernel were not inclined to evict the page anyway.

Table 1, Table 2 and Table 3 show the various states and transitions associated with page-in, page-out and memory access. The first column labels the page’s initial state, the value in a particular column shows the new state resulting from the action listed in the top row with footnotes indicating any messages sent as a result. An entry of “n.a.” means that the situation doesn’t occur; “ignore” means that the kernel ignores the manager’s message.

Table 1 Page-in Transitions

initial state	data supply[a]	data unavailable[b]	data error[c]
out	ignore	ignore	ignore
fetch	$\langle\overline{m},a,a,x\rangle$[d]	$\langle\overline{m},w,w,\overline{p}\rangle$[e]	out[f]
$\langle -,-,-,- \rangle$	ignore	ignore	ignore

a. The manager uses **memory_object_data_supply** to provide the page. The manager can choose the access (*current* and *desired* accesses are set to be the same) and precious flag.
b. The manager uses **memory_object_data_unavailable** to have the kernel supply the page.
c. The manager uses **memory_object_data_error** to indicate that it doesn’t have the page.
d. All stalled threads reattempt their accesses.
e. All stalled threads reattempt their accesses.
f. All stalled threads take memory access exceptions.

Table 2 Page-out Transitions

initial state	page-out[a]
out	n.a.
fetch	n.a.
$\langle\overline{m},-,-,\overline{p}\rangle$	out[b]
$\langle\overline{m},-,-,p\rangle$	out[c]
$\langle m,-,-,- \rangle$	out[d]

a. The kernel decided to evict the page.

b. The page is discarded.
c. The page is returned (**memory_object_data_return**).
d. The page is returned (**memory_object_data_return**).

Table 3 Memory Access Transitions

initial state	peek[a]	poke[b]
out	fetch[c]	fetch[d]
fetch	fetch	fetch
<–,n,n,–>	<–,n,r,–>[e]	<–,n,w,–>[f]
<–,n,r,–>	<–,n,r,–>	<–,n,w,–>[g]
<–,n,w,–>	<–,n,w,–>	<–,n,w,–>
<–,r,r,–>	<–,r,r,–>	<–,r,w,–>[h]
<–,r,w,–>	<–,r,w,–>	<–,r,w,–>
<$\overline{\text{m}}$,w,w,–>	<$\overline{\text{m}}$,w,w,–>	<m,w,w,–>
<m,w,w,–>	<m,w,w,–>	<m,w,w,–>

a. Some task attempts to read the page. If the task doesn't have read access, it takes a memory access exception with no effect on the page.
b. Some task attempts to write the page. If the task doesn't have write access, it takes a memory access exception with no effect on the page.
c. The kernel sends **memory_object_data_request** (read access).
d. The kernel sends **memory_object_data_request** (write access).
e. The kernel sends **memory_object_data_unlock** (read access).
f. The kernel sends **memory_object_data_unlock** (write access).
g. The kernel sends **memory_object_data_unlock** (write access).
h. The kernel sends **memory_object_data_unlock** (write access).

Table 4, Table 5 and Table 6 show the transitions resulting from the **memory_object_lock_request** call.

In Table 4, the *should_flush* parameter is not set, so the kernel retains the page image; and the *should_return* parameter is set to return no pages. Note that both the current and desired accesses are set to be the same. Any threads waiting for increased access re-attempt their accesses.

In Table 5, *should_return* is set to return dirty (modified) pages (MEMORY_OBJECT_RETURN_DIRTY) or both dirty and precious pages (MEMORY_OBJECT_RETURN_ALL), but *should_flush* is still not set. The important point here is that the modified flag is cleared in all cases.

Table 4 Memory "Lock" Operation Transitions

initial state	null access	read access	write access
out	ignore	ignore	ignore
fetch	ignore	ignore	ignore
$\langle -,-,-,- \rangle$	$\langle -,n,n,- \rangle$	$\langle -,r,r,- \rangle$	$\langle -,w,w,- \rangle$

Table 5 Memory "Clean" Operation Transitions

initial state[a]	null access	read access	write access
out	ignore	ignore	ignore
fetch	ignore	ignore	ignore
$\langle \overline{m},-,-,- \rangle$	$\langle \overline{m},n,n,- \rangle$	$\langle \overline{m},r,r,- \rangle$	$\langle \overline{m},w,w,- \rangle$
$\langle m,-,-,- \rangle$	$\langle \overline{m},n,n,- \rangle$	$\langle \overline{m},r,r,- \rangle$	$\langle \overline{m},w,w,- \rangle$

a. If *should_return* is MEMORY_OBJECT_RETURN_ALL, both dirty and precious pages are returned.

In Table 6, *should_flush* is set, so the kernel does not keep the page. As such, specified access modes are ignored. Transitions to *out* indicate that the kernel discards the page.

Table 6 Memory "Flush" Operation Transitions

initial state	clean none	clean dirty	clean all
out	ignore	ignore	ignore
fetch	ignore	ignore	ignore
$\langle \overline{m},-,-,\overline{p} \rangle$	out	out	out
$\langle \overline{m},-,-,p \rangle$	out	out	evict
$\langle m,-,-,- \rangle$	out	evict	evict

Virtual Copy Optimizations

There are two situations in the MK++ 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_map** is called with the *copy* flag set.

These situations occur frequently. Copy-inherited memory is used to support the POSIX **fork** semantics for new process creation. **vm_map** copy supports the POSIX **exec** semantics.

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

- Some of the data is not actually referenced.
- Neither task modifies some of the data, so they can effectively share the same memory image of the unmodified data.

The memory object resulting from a virtual copy is always an object managed by the default memory manager.

On the one hand, these optimizations are purely internal and therefore not part of the MK++ 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.

When a range of memory is to be copied, the kernel creates a new internal memory object to represent the virtually copied pages. This new memory object (the copy destination) may well share actual physical memory pages with the old memory 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 source memory object, as discussed below.

The typical memory object has the property that its pages can be modified only 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 is 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 unmodified 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).

CHAPTER 5 Kernel Control

The MK++ 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.

Ledgers

Some MK++ 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 consumption of 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 resources. Each ledger limits two types of kernel resources:

- Amount of wired kernel memory (data structures, wired pages)
- Amount of (temporary) pageable memory (memory backed by the default memory manager); this is actually a limit on the size of the address space

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 name port is required in the **memory_object_establish**, **alarm_create** and **device_open** calls. These objects (memory cache objects, alarms and devices) only consume kernel wired memory; they do not consume default memory manager space.

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 equivalent), a ledger must be supplied. The resources consumed by threads are considered to be associated with their containing task.

The task that initially holds the receive right for a port (the target of **mach_port_allocate**) is the source from which the port (including the queued messages) is, by default, drawn. (The resource ownership of a port is not changed when its receive right moves.) A port can be created drawn against a specific ledger with *task*, *ledger_name* → **ledger_create_port**.

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.

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

A ledger is created via *host_control* → **ledger_create**. An initial ledger (with infinite limits) is created at host boot time, from which the bootstrap task draws its resources.

Processor Set Control

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

Any given processor set supports the scheduling of its threads according to the scheduling policies implemented for that processor set. 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.

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.

Since some of the implemented scheduling policies permit threads to over-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_info**. 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).

When a task/thread 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**.

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).

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

The list of all tasks assigned to a processor set can be found with *processor_set_control* → **processor_set_tasks**.

Clock Control

Each clock provided by the clock facility is named by a control port (obtained by *host_control* → **host_get_clock_control**) that is used only to set the time or resolution of the clock. A clock supports the following operations:

- *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.

Host Control

A host has its own physical memory pool, set of devices, default memory manager, threads and tasks, and clocks. Whereas many of these objects can be manipulated by random tasks, the host as a whole is typically manipulated only by privileged tasks.

Each host 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 later in this chapter.

Figure 7 shows the client visible host structures.

The following host privilege calls are provided:

- *host_control* → **host_get_boot_info** — Return operator supplied boot-time information.
- *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.
- *thread* (*host_control*) → **thread_wire** — Marks the thread as resource "wired"—always eligible to be scheduled and allowed to consume memory even when free memory is scarce. This property is assigned to threads within the default page-out path.

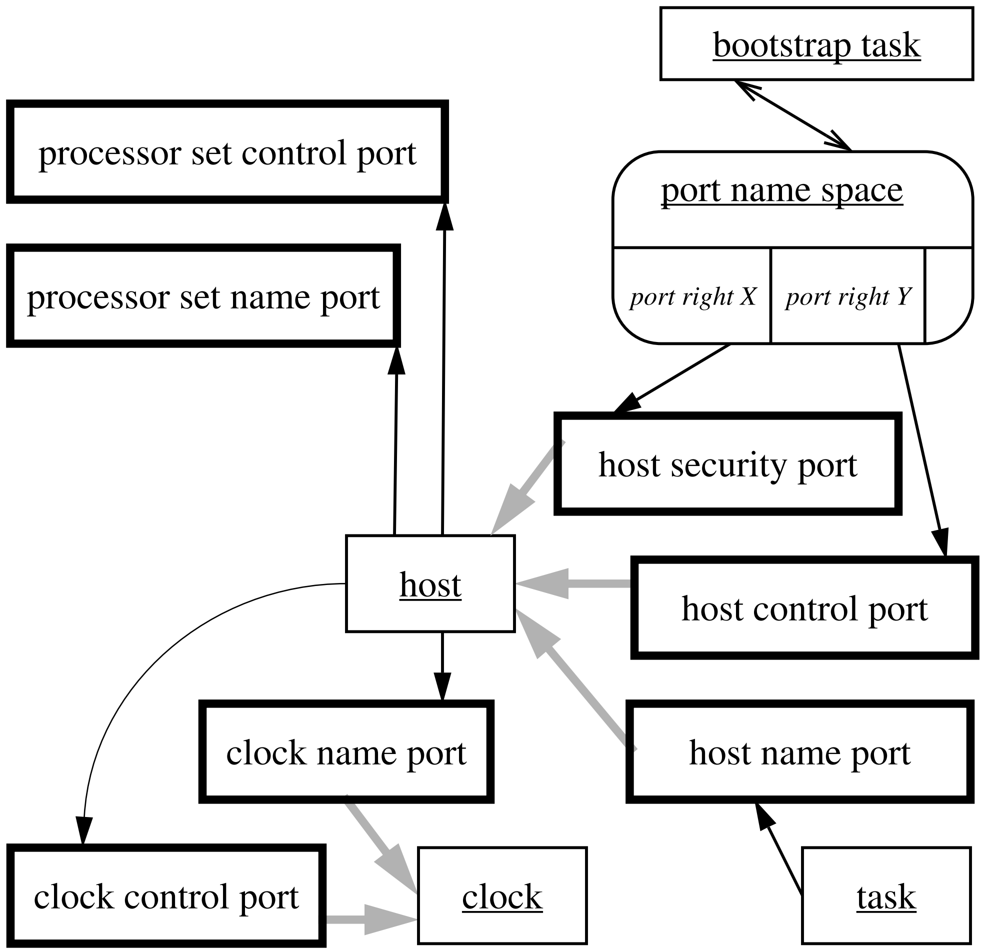


Figure 7 **Host (and Clock) Structures**

Once initialized, a MK++ 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 MK++ kernel itself maintains no permanent state, the subsequent reinitialization of a MK++ 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 of 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.

Host Security

All tasks are tagged with a security ID, an opaque (from the kernel's point of view) identifier that encodes the identity and security attributes of the task. All messages sent by a task are tagged by the kernel with this unforgeable security ID so that servers can reliably determine the identity of the sender of a message for use in making access control decisions.

By default, a task inherits the security ID of its parent. Since this ID is the task's identity, privilege is required to create a task with a different ID. The **task_create_security_id** call permits a task presenting the host *security* port to create a task with an arbitrary identity. No primitive returns the host security port; it is supplied to the bootstrap task as described later in this chapter.

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

The set of active security IDs (those for which there is an existing task) can be found with *host_security* → **host_security_id_list**.

Kernel Device Master

When initialized, the MK++ kernel builds an internal table that lists each device. It exports a single port, the *device master* port, that 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*.

Given access to the device master port, the port controlling a named device is returned by *device_master*, *ledger_name* → **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) and allows operations on the port according to 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 later in this chapter. Figure 8 shows the visible device structures.

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**, virtual copies, etc.). Since no memory manager can provide paging for it, its data writing path is completely

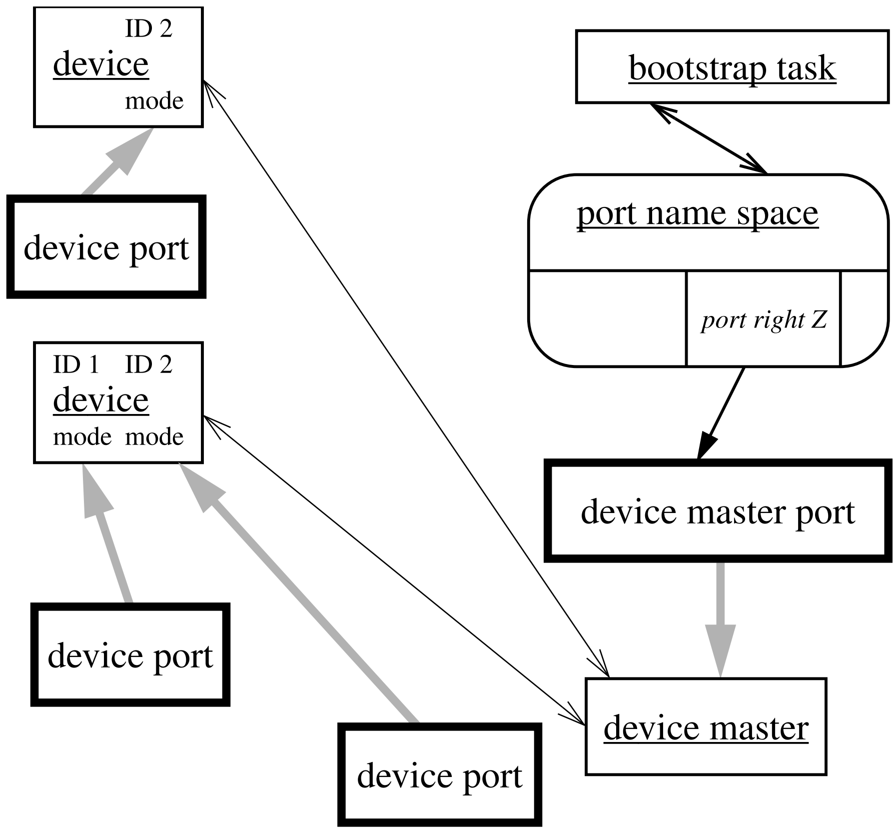


Figure 8 **Kernel Device Structures**

wired in memory; 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.

Unlike other memory managers, the default memory manager sees only one paging object: the abstract paging object. The kernel manages the logical content of this object, simply requesting pages by index from it and returning pages to it (not necessarily from where they were obtained).

The default memory manager communicates with the kernel via the *virtual paging* object, whose name (port) is supplied to the system bootstrap task (and subsequently provided to the default memory manager). The virtual paging object supports two operations:

- *virtual_paging* → **vm_set_default_memory_manager** — Announce to the kernel the abstract backing space port, a port held by the default memory manager to which the kernel will send paging requests.
- *virtual-paging* → **vm_set_recycle_limit** — Put a suggested upper bound on how far into the abstract paging file the kernel should write before recycling space.

Being a potentially replaceable user space entity, the default memory manager can export services of its own. The default manager currently exports only one service interface, *default-pager* → **add_partition**, which is how (privileged) system servers announce to it what device partitions should be used for backing space.

Bootstrap Task

The bootstrap task is created with all of the privileges that the kernel exports, including access to the host control, host security 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.
- Ledger — an initial infinite-limit ledger.
- Security ID — KERNEL_SECURITY_ID.

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 access modes that are necessary for their functioning.

When the bootstrap task is created, its bootstrap port is set to be a special kernel port (one whose receiver is the kernel). The bootstrap port responds to only one request for service. One of the first things that the bootstrap task does when it starts is to send a message to this port (*bootstrap* → **bootstrap_ports**). The reply from this request contains the privileged port rights: the host control port, host security port, device master port and the virtual paging object 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.

CHAPTER 6 Design Strategy

This chapter gives an overview of the internal structure of the MK++ kernel as an aid to understanding its operation.

The MK++ kernel supports a variety of abstractions that are further decomposed within the kernel. Although the semantics associated with these abstractions are rich, they still represent a minimal set of abstractions in that each is needed to support some TCB or client service in a performant way.

The kernel is internally divided into a number of subsystems, each completely contained within some layer, and those subsystems are further divided into modules (objects and associated methods). The layers of the kernel are true hierarchically structured layers; i.e., there are no upward dependencies. Each layer provides one or a few basic abstractions that are briefly presented in the description of each layer. Modularity hides the implementation details of each abstraction, with each module providing a well defined set of operations for the abstraction that the module implements.

Object-oriented design allows for a natural decomposition of abstractions in accordance with the requirements of layering. Most of the kernel objects visible at the kernel interface are actually decomposed into various layers of abstraction, where each layer derives from lower layers via the use of object inheritance. For example, the notion of “thread” starts with the Processor Scheduling defined notion of “execution point”, from which the Transfer Management layer derives the notion of “transfer receiver” (IPC synchronized thread) and so on up, until the user visible notion of thread is derived.

Multiple inheritance is extensively used to allow multiple disjoint paths of abstraction. For example, “thread” derives both from “kernel object,” which defines what it means to be a user visible kernel object capable of being the target of message invocations and also “transfer receiver,” which defines what it means to be schedulable and to be able to synchronize with other threads. The use of multiple inheritance permits the logical separation of different abstractions that are combined only when necessary.

Since all of the subsystems within the kernel execute within the same (protected) address space, they, in principle, can access each other's data and procedures. However, being of object-oriented construction, the data objects within the kernel are encapsulated within language defined modules, with only those modules capable of manipulating the internal state of the data objects. Language features are further used to restrict access to the various methods defined by those modules to only those modules needing access in accordance with the principle of least privilege.

The use of IPC to invoke kernel functions centralizes the kernel invocation, basic parameter validation, and kernel object access methods. Kernel objects themselves are insensitive to the mechanics of invocation.

Hardware dependencies are carefully hidden inside machine independent abstractions. For example, the kernel is insensitive to whether the underlying hardware's virtual memory support involves page tables, inverse page tables or direct translation buffers; these details are hidden in the "address translation map" module within the Resident Memory Management layer. Access to privileged hardware functions are also encapsulated and use is restricted in accordance with least privilege.

Glossary

Some of the entities that appear to be the same at the kernel interface are quite different within the kernel, and vice versa. Before identifying the subsystems within the kernel, the following internal kernel notions need to be understood.

Ports

What a user task thinks of as a port (right) can name one of two things: a user port, an object whose receive right is held by a user task; or a kernel object, an object to which a sent message is interpreted as an invocation. Within the kernel, the common aspects of both of these two user named objects is the *connection* class. It is important to note that kernel objects don't actually have port objects naming them. User ports, on the other hand, are implemented by user port objects.

Kernel Object

A kernel object is a kernel entity which is accessible outside the kernel via kernel rights.

Port Identity

Ports (actually, connections) within the kernel are identified in a way different from the way they are identified outside of the kernel.

- Port name — user space local value that names a port right
- Port right — kernel managed capability that names a port via a port locator

- Port lock — data structure that physically locates connection objects

Resource

The kernel's resource management interfaces are associated with ledgers. Internal to the kernel, a ledger identifies a resource account that keeps track of the amount of resource usage as well as the *resources* bound to that account. A *resource* is a kernel entity whose space consumption is individually tracked. Each connection object represents a resource. All other data structures are considered to be "owned" by some resource.

Events

The kernel makes a big distinction between internal and external events.

An internal event is one that the kernel causes to be satisfied. A thread executing in kernel context that is waiting for such an event is not interrupted or terminated; it waits for the event to be satisfied.

External events, on the other hand, are events that user tasks cause to be satisfied. A thread waiting for an external event can be interrupted and terminated (once it cleans up after itself). There are two sets of external events: waiting for a message and waiting for an external memory manager to supply a page.

Device interrupts correspond neither to internal nor external events. A thread waiting for a device operation to complete is actually waiting for a message. The completion of a device operation causes the kernel to send a message and to clean up the associated operation, but the kernel must be prepared for the device operation never to complete.

Kernel Subsystems

Figure 9 shows the internal structure of the kernel. A horizontal bar in the diagram indicates a layer within the design. Most of the layers consist of a single subsystem.

Processor Scheduling

The Processor Scheduling subsystem, the lowest subsystem in the kernel structure, is responsible for controlling the context in which each processor is executing at any moment and what activity is being performed on that processor. A processor may execute code associated with some logical execution point (the abstraction, upon which threads are built, representing their ability to be scheduled), or it may execute code associated with an interrupt handler.

The Processor Scheduling subsystem implements all transitions between processor execution contexts and between execution points, either by direction of hardware or software. The machine-dependent mechanics involved with these transitions (saving and restoring registers, identifying the hardware address translation tables to use, manipulating interrupt priority level, etc.) are localized in this subsystem.

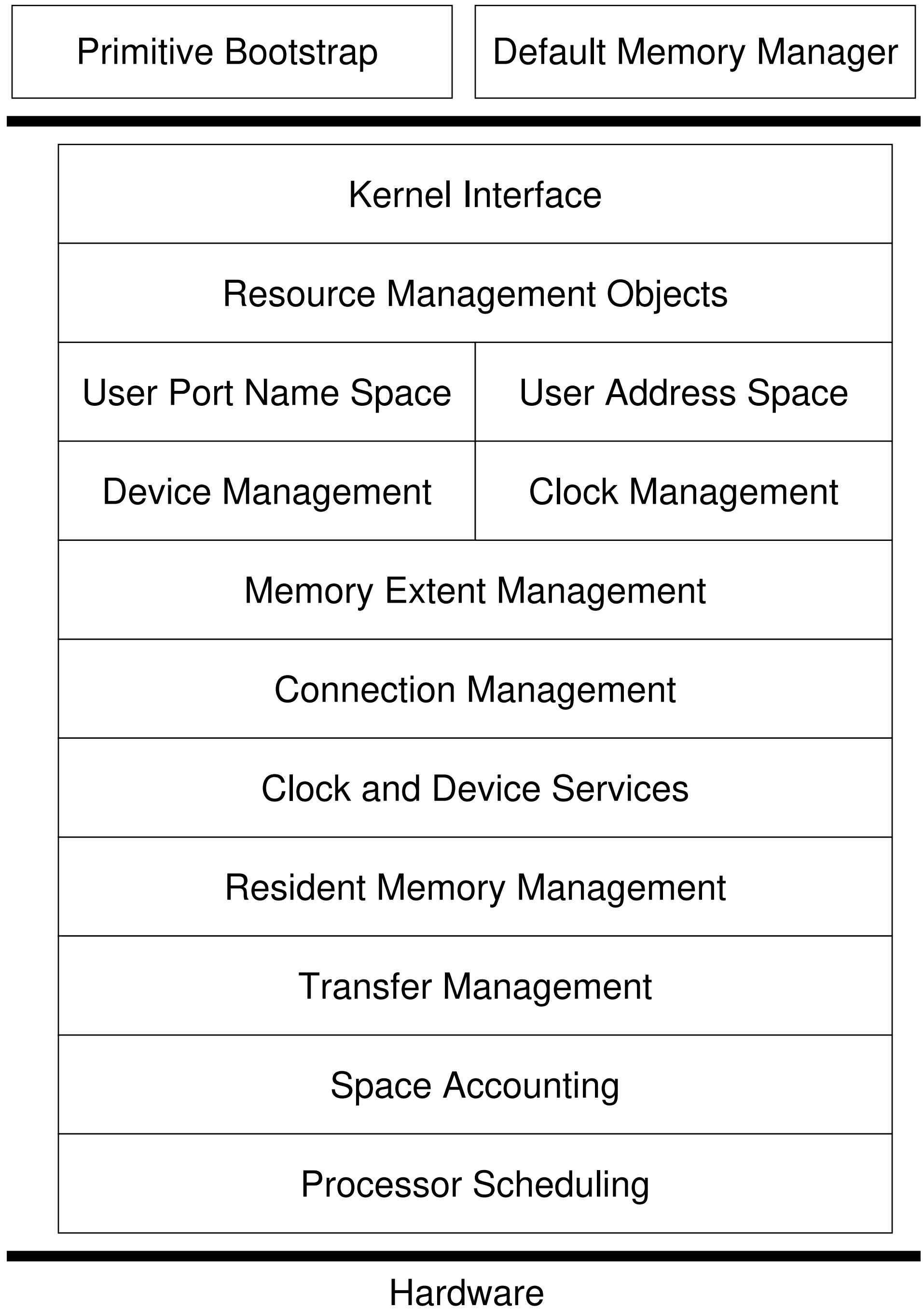


Figure 9 **Internal Kernel Structure**

The facilities within this subsystem are as follows:

- Execution point scheduling
- Hardware trap/interrupt vectoring
- Synchronization services
- Hardware thread state management

Space Accounting

One aspect of the kernel's resource management is the tracking of kernel memory (space) consumption so that higher layer resource management mechanisms can determine if kernel memory (one aspect of physical resources) is being used excessively. Space Accounting keeps track of kernel space consumption.

With the exception of certain static data structures, every data structure within the kernel is either associated with a space source, is itself a space source, or is derived from a space source. A space source keeps track of the amount of kernel memory it and its associated structures consume. Each user port and kernel object has an associated space source.

A space account keeps track of the total consumption of all space sources associated with it. Each ledger has an associated space account.

Transfer Management

The Transfer Management subsystem provides the mechanisms to enqueue, dequeue or directly transfer items to some transfer receiver (some execution point). A *transfer item* is an abstraction representing something of interest to the transfer receiver. The most obvious example of a transfer item is an IPC message but the abstraction is also used to represent work to be done by something other than the sender.

This subsystem presents objects possessing the ability to send and receive transfer items. Most of these objects are used in various ways by Connection Management to support the user IPC service, but they are also used purely internally to transfer work for deferred processing by other kernel threads.

Resident Memory Management

The Resident Memory Management subsystem is responsible for managing the use of the hardware's resident physical memory. The major activity in this regard is the paging of pageable virtual memory. This subsystem owns the page residency (replacement) policy and implements the page-based external memory management protocols.

As part of managing resident virtual memory, this subsystem implements machine-dependent address translation operations for address spaces. This subsystem provides an address translation map object for each address space. This object encapsulates the machine-dependent data structures that are used by the address translation hardware.

This subsystem also provides kernel resident memory management services: allocation and deallocation as well as support for managing the use of kernel memory to support user initiated kernel operations.

Clock and Device Services

The Clock and Device Services subsystem performs I/O, providing controller and device-specific data transfers. This subsystem also manages the hardware clocks, providing time recording and announcement (alarm) services. The handling of all device interrupts is performed within this subsystem.

Connection Management

A *capability* is a globally unique name that maps to a communication endpoint. Every distinct capability target is named by a distinct capability. A capability value is never reused, even after the target object is destroyed. A capability is a name that can always be safely used to refer to the target, although attempting to reference the target may return an error. The basis for an IPC port right is a capability.

A *connection* is the abstract relationship between a capability and some target. There are two types of connections: those representing user ports and those representing first-class kernel objects such as ledgers, tasks, and threads.

The Connection Management subsystem provides the system-wide view of IPC. In general, connection management operations map a capability naming a connection to its target and invokes the target directly. In some circumstances, the subsystem maps the capability to a capability target, one of a class of objects that represent the ability to send into, receive from, or in other ways operate upon, the logical target of the capability or the connection itself.

The ability to name a first-class kernel object by a capability is an important function of this subsystem; all first-class kernel objects find their basis here.

As part of managing these connections — in particular, connection life and death — Connection Management implements the “dead name” notification service.

Every kernel entity whose lifetime is under user control (a billable resource) is associated with a connection, be it a user port or any of the many classes of kernel objects. Connection Management tracks the relationship between connections and resource accounts, objects provided so that higher layer resource management mechanisms can locate all connections (resources) associated with a particular account.

Finally, the ability to group multiple user port connections into a port set is provided here.

Memory Extent Management

The Memory Extent Management subsystem provides the system-wide view of virtual memory — how memory is identified or named and the relationship of one piece of memory to another.

Named memory, which is managed by a memory manager other than the default memory manager, is named by the *memory_cache* object. This object implements the memory object initialization and termination protocols.

The management of the relationship of memory to other memory consists of the tracking of memory sharing and virtual copying.

Clock Management

The Clock Management subsystem provides the kernel's time objects abstractions. It associates the user visible clock and alarm objects with their underlying physical clocks and alarm requests.

Device Management

The Device Management subsystem provides the kernel's device object abstraction. It associates the user visible device objects with their underlying physical devices. This subsystem is responsible for naming devices and providing the device open/close services. The activity of generating a memory object representing a device is provided here.

User Address Space Management

The User Address Space Management subsystem manages the task local view of virtual memory, directing operations upon regions of a task's address space to Memory Extent Management. Access control over user memory is enforced within this subsystem.

User Port Name Space

This subsystem is responsible for the management of user port names. It maps user port names to internal capabilities. All operations that concern a task's name for a capability are provided here, including requests for notifications associated with user port names.

Although port sets are implemented by Connection Management, the ability to name them and use them as the target of a message receive operation is handled in this subsystem. It is this subsystem that understands when a task holds a receive right and can operate upon it as such.

Resource Management Objects

The Resource Management Objects subsystem provides the high level resource management facilities for the kernel and implements the management of user-visible ledgers, tasks, and threads.

The management of ledgers consists of providing the handle for resource creation and destruction (both the user-visible ledger and the underlying resource account) and implementing resource reclamation when the ledger is destroyed.

A special facility of this subsystem is implementing kernel space management policies.

Closely associated with tasks, threads, and ledgers are objects that provide some level of control over these objects: the host and processor sets. The Resource Management Objects subsystem also provides these objects.

A significant part of this subsystem's provision of user threads are the high level thread management primitives such as terminate and suspend.

Kernel Interface

The Kernel Interface subsystem defines the boundary between kernel thread execution and user thread execution. Via mechanisms provided by the Processor Scheduling subsystem, all entry into the kernel by user mode tasks is into the Kernel Interface subsystem, and it is from this subsystem that the thread exits. It is the top of the kernel — all user-initiated requests originate here.

Primitive Bootstrap Task

The Primitive Bootstrap Task is a user mode task created by the kernel initialization process. The purpose of this user mode task is to create, load, and start the Default Memory Manager and the other trusted MK++ tasks.

Default Memory Manager

The Default Memory Manager is a user mode task (actually, part of the bootstrap task) that provides backing storage for anonymous regions of virtual memory by maintaining the system's paging store object. This object represents the disk partitions used to store paged-out anonymous memory.

CHAPTER 7 Security Constraints

The MK++ kernel enforces various security constraints of the overall system. Being dependent only on the hardware, there are no assumed constraints for the kernel. The enforced constraints are the following:

- The kernel applies the correct message tag to each IPC message sent from one task to another.
- The kernel applies the kernel's message tag to all messages it sends.
- The kernel permits a task to access a port only if the task holds a port right to that port.
- The kernel does not allow a task to map into its address space a memory object via a memory object representative that does not represent the identity of that task.
- The kernel does not allow access to a memory object in an access mode not allowed by the memory object representative through which access to the memory object was obtained.
- The kernel rejects any device operation via a device port that does not represent the identity of the requesting task.
- The kernel rejects any device operation that implies an access mode not permitted by the device port through which the device is being accessed.
- The kernel's device drivers maintain the integrity of data on their associated storage devices.