MK++ Kernel
Executive Summary

Version 2.0 — PRELIMINARY

June 12, 1997

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
Version 2.0	June 12, 1997	Documents MK++ Release 2.0

Copyright 1993-1997 by The Open Group, Inc.

All rights reserved.

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

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

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

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

Contents

CHAPTER 1 Introduction 5
Kernel Abstractions 6
Architectural Elements 8
Kernel and Object Invocation 9
Document Conventions 10

CHAPTER 2 Inter-Component Transactions 13

CHAPTER 3 General MK++ Manipulations 15
Threads 15
Tasks 19
RPC Service and Related Abstractions 21
RPC Payload Contents 24
Empty Threads 24
Thread Migration and Transfer of Control 25
Port Name Space Semantics 27
Port Pool Semantics 28
The RPC Kernel Trap Interface 29
Virtual Address Spaces 31
Kernel Devices 33
Clocks 34
Alarms 35
Processor Sets 35
Hosts 36
Synchronizers 37

CHAPTER 4 External Memory Management 39
Basic Manipulation 39
Memory Object Initialization 40
Basic Page Manipulation 44
Memory Object Termination 44
Precious Pages 45
Direct Cache Manipulation 45
Summary of Page Manipulations 46
Virtual Copy Optimizations 49

CHAPTER 5 Kernel Control 53
Ledgers 53
Processor Set Control 54
Clock Control 55
Host Control 55
Host Security 56
Kernel Device Master 57
Default Memory Manager 57
Bootstrap Task 58

CHAPTER 6 Security Constraints 61

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 a Remote Procedure Call (RPC) service that allows separate computational entities to transmit data between themselves. The goals of MK++ include:

- Exploiting parallelism in both operating system and user applications.
- Supporting large, potentially sparse address spaces with flexible memory sharing.
- 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 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 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.
- Payload — A collection of data objects transferred between computational entities via the RPC mechanism. The kernel appends an unforgeable security tag to each payload that identifies the sender; the security tag mechanism can be used by service providers to control access to the objects they manage.
- 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”.

Tasks are relatively expensive entities. All of the threads in a task share all resources contained in the task. 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 lightweight 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 addresses 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; the kernel interface exports abstractions that 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 central to MK++'s programming model. This model is especially suited to the development of client/server systems in which client tasks access services by invoking RPCs (Remote Procedure Calls) on objects managed by server tasks. Since the MK++ kernel provides few services of its own (for example, it provides no file service), an MK++ task must communicate with other tasks to access traditional operating system services.

RPCs are transmitted between tasks via communication channels are called *ports*. 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 handle RPCs invoked on the port. Multiple tasks can hold *send rights* to the port; a send right may be used to invoke RPCs via the port.

A task can acquire a port right only via one of two ways:

- The task receives the right from another task or the kernel via an RPC.
- 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.

Architectural Elements

The MK++ programming environment provides access to the following operating system abstractions:

- thread — An execution point.
- 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 pool — A set of ports whose RPCs are to be handled by the same routine.
- 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 pool.
- payload — A collection of data transmitted from one task to another. A particular payload may be used to transmit port rights to its target. The kernel tags each payload with an unforgeable security tag that the receiver may check to verify the sender's identity.
- notification port — A port established by a user space entity to receive remote procedure calls from the kernel. This is the means by which the kernel notifies a task that a particular I/O operation has completed, or that a particular port has been deallocated.
- 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 notifications 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 notifications 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 create and destroy these entities, and to manipulate their state.

The kernel is just one manager that can implement resources 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 via the kernel's RPC service. The service to be provided is determined by the manager that handles RPCs targeting 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 managing that entity.

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.

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 via RPCs; 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 are initiated by the invocation of an RPC on a port representing the object to be operated upon. In this document, this is shown in the form:

object → **function**

which means that the **function** is to be invoked via an RPC that target's the port representing the object. 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, an RPC is issued against the containing virtual address space (named by the owning task). To manipulate a port name/right (and often, the associated port), an RPC is issued against the containing port name space (named by the owning task). A subscript notation,

object [*id*] → **function**

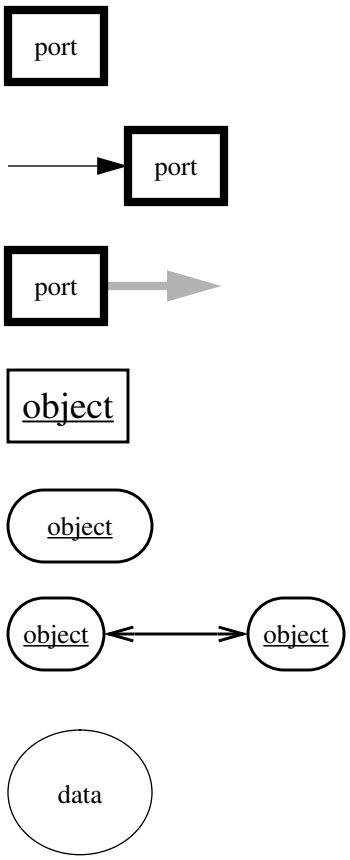
is used to show that an *id* is required as a parameter in the RPC to indicate which range or element of *object* is to be manipulated.

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 RPC 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 the data contained in an RPC payload.

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.

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 **mk_rpc** trap, by which most kernel invocations are made (as well as service invocations of other tasks). If a given invocation of **mk_rpc** trap targets a port representing an object exported by the kernel, the trap constitutes a kernel service request (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 **mk_rpc** trap does not return until the kernel has performed the desired operation and returned the results.

The kernel also provides a notification service that allows user space tasks to request kernel-triggered event notifications.

The following list classifies all supported inter-component transactions.

- 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 external memory management notification to some TCB or non-TCB client.
- Kernel sending a default memory management notification 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 notification to some TCB or non-TCB client.
- Kernel sending an exception RPC to some TCB or non-TCB client.
- Kernel sending an RPC notification to some TCB or non-TCB client.
- Kernel sending a network device filter packet notification to some TCB or non-TCB client.
- Kernel sending a program counter (PC) sample RPC 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 lightweight 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 invoke and RPC on behalf of the thread.

Figure 2 shows the client visible structure associated with a thread. The thread object is the receiver for RPCs invoked via 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.

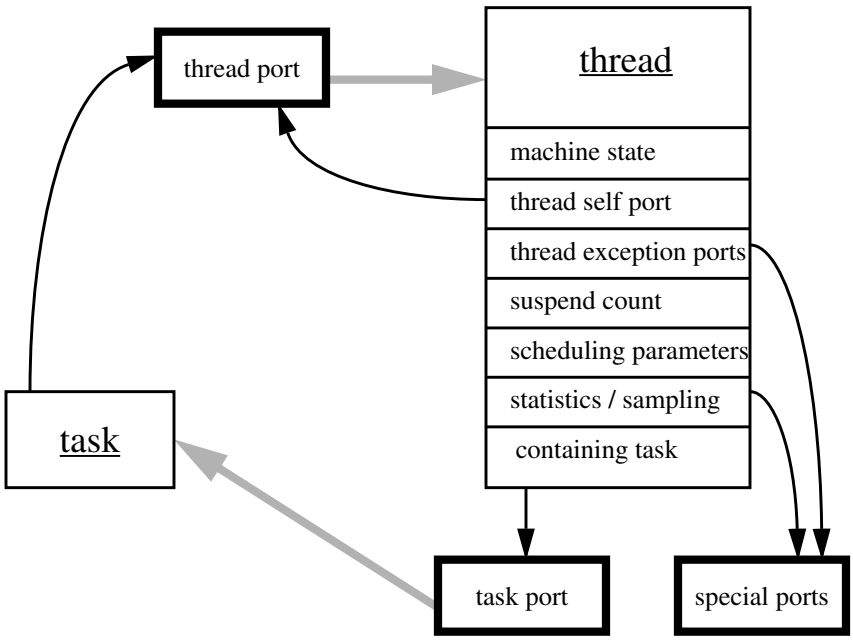


Figure 2 **Thread Structures**

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 and invoke RPCs against them; the various things it can do are discussed under the sections describing the object manipulated.

Note that the MK++ traps described below are functional 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** RPCs. 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).

RPC Traps

Access to the RPC facility is implemented via three traps: **mk_rpc**, **mk_rpc_reply**, and **mk_rpc_reply_buff**.

The **mk_rpc** trap invokes a synchronous RPC against a caller-specified port. Data to be transmitted with the RPC is contained in a caller-sepcified buffer and may contain port rights and addresses of regions of separately allocated memory, as well as untyped user data.

The **mk_rpc_reply** trap is called by a server to return control to the invoking thread. The data transmitted with this reply is contained in a reply buffer. While this call does not normally return, if the reply buffer's data is missing or is mis-protected, the trap returns the following error: MACH_RPC_INVALID_DATA.

The **mk_rpc_reply_buff** trap is similar to **mk_rpc_reply**, but allows the server to install a new buffer for receipt of data for the next incoming RPC so that the current buffer's contents need not be copied out immediately, but the server is responsible deallocating the current buffer at some point. Like **mk_rpc_reply**, this call does not return except to indicate that the reply buffer is incorrect, with MACH_RPC_INVALID_DATA.

Exception Processing

When a thread generates an exception, it is trapped into the kernel; while executing in kernel context, the thread invokes an RPC on an exception port. The RPC's contents describe the nature of the exception.

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 RPC. If neither port is defined, the thread is terminated.

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

- *exception_port* → **catch_exception_raise_state** — Indicates the nature of the exception and includes status codes and the appropriate flavor of thread state.
- *exception_port* → **catch_exception_raise_state_identity** — In addition to specifying the nature of the generated exception, this exception form identifies the thread and its task by including the task and thread self ports.

In handling a given exception, the handling task may explicitly modify the thread's user register state by invoking *thread* → **thread_set_state**.

Not every exceptional condition that a thread encounters is handled as described above. For example, a page-not-resident fault does not generate an RPC against the exception port; instead, a notification 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 suspend the target thread's execution in user mode. (If the target thread is executing in kernel mode, it is suspended during the transition from kernel mode to user mode.)
- *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.

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 kernel manipulates a task object's state in response to RPCs invoked against the 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.

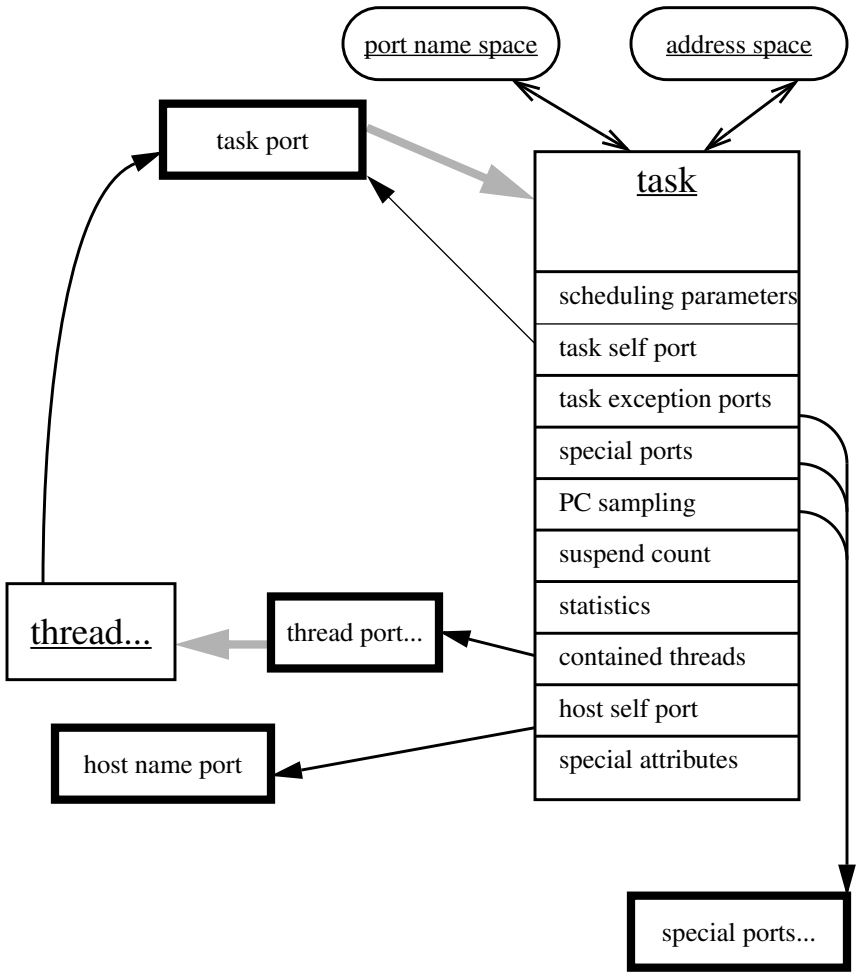


Figure 3 **Task Structures**

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:

- 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 RPC payloads 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**).

RPC Service and Related Abstractions

RPC is the unifying abstraction used for all transfers of data and control between user tasks and between the kernel and a user task. RPC is the mechanism by which servers provide services to clients. The client requesting the service is said to invoke the RPC. An RPC invocation is synchronous, i.e. the client blocks until the server completely processes the request, explicitly returning control to the client.

Ports and Port Pools

A port is a unidirectional communication channel between a client requesting a service and a server that provides it. A port has a single receiver and (potentially) multiple senders. A port that represents a kernel-supported resource has the kernel as the receiver. A port that names a service provided by a task has that task as the port's receiver.

Note that a port by itself does not have sufficient resources to provide the service it abstracts. A port must belong to a port pool before clients can access the services that port represents. A port pool is the mechanism by which a server provides the services abstracted by one or more ports; it allows a set of ports to be treated as a single unit when servicing RPC invocations. RPC reception is allowed only against a port name that names a port pool.

A port may abstract one of the following services:

- an RPC service
- an exception service (including alerts)
- a notification service: dead name, I/O completion, alarm clock

Ports are created implicitly when any other system entity is created, since kernel objects are represented to the user as send rights. Ports are created explicitly by **port_allocate**, which takes a flavor indicating which type of port to create:

- Common
- Notification

The state associated with a common port is:

- quality of service attributes
- a list of threads requesting RPC or exception service (a holding area used when the port is not in a port pool)
- a scalar value to be provided to the recipient when an RPC is invoked on the port, in lieu of the true port name

The existence of ports is of obvious importance to all involved. As such, many tasks using a port may wish to be notified when the port is destroyed. A request for such a notification is made with **port_request_notification**.

Notification Ports

Notifications are a special type of RPC invoked by the kernel to notify the user that a particular kernel-initiated event has transpired. The kernel can notify the user about various types of events, but only at the user's request. Requests for notification are registered with the kernel by specifying a port to be invoked when the event occurs. These events include:

- dead-name notifications
- asynchronous I/O completion notifications
- clock alarms

Notification ports are a superset of user ports. Notification ports have a queue for notifications awaiting delivery. Notification ports maintain a list of one or more delivery engines that deliver the queued notifications by invoking an RPC on the underlying port. By default one delivery engine is created when the notification port is created. Additional delivery engines may be allocated and deleted by calling **port_set_attributes**.

In addition to the state associated with a common port, a notification port also has:

- a queue of notification requests
- list of delivery engines for delivering notifications

Dead-Name and I/O Completion Notifications

A dead-name notification does not provide its own shuttle. Hence a notification type of port must be allocated, which has delivery engines associated with it to provide shuttles. This port is then used as the notify port in **port_request_notification**, which will return an error if the port is not the correct type.

This allows a single port to be used to receive notifications for multiple target ports; since dead-name notifications already identify the defunct port, this is easily done.

I/O completion notifications are handled analogously to dead-name notifications. The target port is passed as the reply port in a "request" version of a device service, which will return an error if the port is not a notification port.

To enable a single port to be used to receive notifications for multiple I/O completions, the notifications delivered contain an opaque scalar that identifies which I/O completed. This opaque scalar is supplied in the original I/O request.

Port Rights and Port Name Spaces

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 two types of port rights:

- receive right -- Allows the holder to receive RPCs on the associated port.
- send right -- Allows the holder to invoke RPCs on the associated port.

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. Receive rights cannot be moved between tasks. Send rights can only be copied, from either a send or receive right, and may be passed between tasks.

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:

PORT_NULL -- No associated port right.

A port right -- A send right, receive right, or a send-receive right for a port.

A port pool name -- A name that allows a server to provide resources for RPC service for one or more ports.

mk_rpc is one of the principal ways that rights are manipulated. A send right can be copied from one task to another (a copy of the sender's right is added to the receiver) via RPC. Other than RPC operations, port rights can be manipulated only as members of a port name space. In addition to using **mk_rpc** to generate a send right from an existing send or receive right, send rights can also be forcefully copied by **port_extract_right** (the equivalent of the target sending the right in an RPC) and **port_insert_right** (the equivalent of the target receiving the right in an RPC).

Port rights are created implicitly when any other system entity is created, since kernel objects are represented to the user as send rights. Port rights are created explicitly by **port_allocate**. A port right is destroyed explicitly by **port_destroy**. Destruction can also be a by-product of port name space manipulations, such as **port_mod_refs**. Some status information can be obtained, given a right, with **port_get_attributes**.

Port Resources and Attributes

A port has its kernel resources drawn from the target task's ledger. A port is destroyed only when its receive right or its owning ledger is destroyed. When a port is destroyed, it is implicitly removed from any port pool to which it belongs. Any pending notifications are destroyed, as are any notifications that are triggered after the destruction of

the requesting port. Any pending RPC or exception invocations return an error. Any associated delivery engines are deleted.

Most of the attributes of a port are assigned at the time of creation. Quality of service attributes can be changed by **port_set_attributes**. For example, ports abstracting a notification service are created with one delivery engine to implement the service, allowing single-threaded notification delivery. The number of concurrent notification deliveries may be increased using **port_set_attributes**, which implicitly allocates more delivery engines. Note that the number of empty threads in the port pool needs to be at least as great as the number of delivery engines in order to take advantage of additional concurrency, since one of each is required to complete the notification delivery. Port attributes can be obtained by **port_get_attributes**.

RPC Payload Contents

Both pure data and port rights can be passed in an RPC invocation and/or reply. The data passed may include untyped data bits and out-of-line (OOL) memory. The passing of port rights is very significant. It is by receiving them in an RPC that a task obtains new rights. (A task can also receive new rights through the **port_insert_right** and **port_extract_right** calls against a task port).

An RPC also includes attributes implicitly added by the kernel. One of these attributes is the invoker's security ID, which allows RPC servers to ascertain the identity of the invoker. To this end, the kernel supplies the security ID of the invoker to the task providing the RPC service. Thus user-initiated RPCs are tagged with the security ID of the invoking user, and kernel-initiated RPCs are tagged with the security ID of the kernel. The kernel deposits the security ID of the invoker at the base of the per-empty-thread stack prior to dispatch, starting the empty thread's stack pointer just beyond the security ID. The run-time is responsible for knowing how to locate the security ID (which is reasonable given that the run-time supplied the stack area). Kernels which do not support security deposit SECURITY_ID_NULL at the base of the empty thread's stack, and are free to do this just once, when the empty thread is created.

Empty Threads

Threads are the abstraction of a single point of control or "thread of execution" within an address space. The execution state of a thread (kernel stack and scheduling parameters) are kept in the thread's *shuttle*, which migrates when performing an RPC. The portion of a thread that does not migrate is known as an *empty* thread when it exists in the absence of a shuttle.

Empty threads are threads that are created with no kernel stack and no scheduling parameters. As such, empty threads cannot execute. An empty thread is the recipient of an RPC invocation. An empty thread is created in a task and associated with a port pool (see **Port Pools** below) by **thread_create_empty**, which also takes parameters indicating the area to be used as a user stack when the thread is invoked, a description of a receive buffer for inbound RPC invocations, and an entry point (a.k.a. dispatch address) which will be called to handle each invocation. Threads created by **thread_create** or **thread_create_running** also include a shuttle, and are said to be *rooted* threads. A thread invoking an RPC donates the necessary resources (its shuttle) to the empty thread so that it is able to execute.

Empty threads belong directly to the task in which they were created and indirectly to the port pool with which they are associated. When the port pool is destroyed, all empty threads associated with it are also destroyed. Empty threads that are awaiting incoming RPCs are destroyed immediately, while empty threads that are currently servicing an RPC are destroyed when they complete the service. Note well that there is no guarantee that the destruction of these threads will be timely, since this depends on the behavior of the thread in question. Explicit thread termination is always guaranteed to work in a timely way. If the port pool is being destroyed as a part of task termination, obviously all threads will be explicitly terminated as part of the tear-down of the task's port name space, so that timely return from an RPC in progress is not an issue.

Support for growing and shrinking the empty thread pool of a server is desirable. Growing the pool is simple but shrinking the pool is a little tricky. For instance, an empty thread can be terminated explicitly, but there is no guarantee that it is not currently servicing an RPC request. A straightforward solution can be implemented entirely in the server without additional kernel mechanism. One strategy is to use a separate resource leveling thread. This thread can be awakened whenever it is time to trim some empty threads from the pool. The leveling thread could invoke a server-specific prune method. That method simply verifies that it really is appropriate to prune at least one empty thread and if so, terminates itself. This will consume one of the empty threads on the server's pool (note that **thread_terminate** only destroys the calling shuttle if the target thread is a rooted thread). The leveling thread can repeat this process whenever desired.

Thread Migration and Transfer of Control

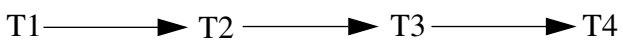
A set of threads connected by means of a sequence of RPCs constitutes a migrating thread. Both user-to-user and kernel-to-user RPCs use this mechanism. Migration effects the transfer of control from the invoking thread requesting a service to the target thread providing the service. To support the migrating thread abstraction, the following resources must be available:

- A rooted thread (user or kernel)
- Empty thread(s) in a port pool, each with:
    - A user stack
    - A user dispatch address

A user thread or delivery engine invokes an RPC on a port. An empty thread from the associated port pool is selected and invoked. An empty thread that has been invoked is said to be *active*. The invoking thread and the active thread form an RPC chain, and both of these are said to be *chained*. The invoking thread donates its shuttle to the empty thread, and the empty thread supplies the user stack and the dispatch address. The formerly empty thread is dispatched to user space to execute the requested service. When the service is complete, the active thread traps back into the kernel where control is returned to the invoking thread. The empty thread then rejoins the port pool.

Since an active thread can subsequently invoke another RPC, RPC chains can be arbitrarily long. RPC chains can also cross node boundaries. The thread currently executing in the RPC chain is said to be the *top* thread or at the *top* of the chain. Any thread in the RPC chain that is not the top thread is said to be *embedded*.

For example, consider the following chain of three RPC invocations:



All four threads are chained. Thread T1 is the rooted thread, thread T4 is the top thread, and threads T1, T2, and T3 are embedded.

Migration of Scheduling Data

Threads involved in an RPC chain are part of a single thread of computation. Therefore scheduling data (policy, priority, etc.) migrates as control passes between threads, so that it continues to influence the computation in different threads. Scheduling information is thus logically part of the shuttle.

In fact, portions of scheduling data need to be stored in the thread as well. To ensure that a scheduling parameter change targeted at a given thread affects threads "above" but not "below" the target in the RPC chain, a snapshot of current scheduling data needs to be stored per-thread. Any scheduling change should affect future RPCs, but should not propagate back to the invoker of the thread at which the change was targeted.

Each chained thread may contain a snapshot of the scheduling information from the shuttle, as of the last time that thread was "top" thread. The per-thread snapshot of scheduling parameters interacts with the "real" (per-shuttle) copy as follows:

- Scheduling parameter changes targeted at an empty thread that is not currently chained are ignored.
- Scheduling changes targeted at a chained thread overwrite the thread's snapshot, and if it is the top thread, the per-shuttle parameters are changed as well.
- When a shuttle returns to a chained thread, that thread's snapshot (if any) is installed in the shuttle.

While the mechanism implementing this behavior may be optimized, the behavior is important in that it allows nested or scoped changes to be made to the scheduling attributes of a thread engaged in arbitrarily long RPC chains, and also affords implicit protection against unwanted changes. By default, a thread operates using its root scheduling attributes, yet a server has complete local control over the scheduling of threads within its domain, and servers in the chain are protected from the effects of unexpected scheduling changes by others.

The interfaces which explicitly modify scheduling attributes are **thread_policy**, **thread_set_policy**, and **thread_depress_abort**; scheduling parameters are implicitly modified by **thread_switch**, **task_policy**, and **task_set_policy**.

To enable a server to restrict incoming RPCs based on their scheduling parameters, i.e., to accept only "compatible" migrating threads, the server may specify what scheduling policies incoming shuttles must have using **port_pool_set_attributes**. Invocation attempts by threads with other scheduling policies return an error.

Exceptions and Alerts

Exceptions are caused by a thread's own execution, and are delivered to the thread's exception port (or, if none exists, the task's exception port). Alerts are a software interrupt mechanism, delivered using a type of exception. In both cases the thread is blocked until the exception delivery is completed. The thread can therefore donate its shuttle to the empty thread servicing the exception. Additionally, there can be only one exception delivery at a time in progress per thread, and while multiple alerts may be targeted at a given thread, there is a separate mechanism for queueing the alerts.

Delivery Engines

RPCs initiated by the kernel require an object that can 1) donate its shuttle to the invoked empty thread performing the RPC service and 2) block if there are no empty threads to perform the service. The object that performs these functions is called a *delivery engine*. A single delivery engine is created when a notification port is created. Additional delivery engines may be associated with a port by calling **port_set_attributes**. Delivery engines are removed when the associated port is destroyed or by calling **port_set_attributes**.

Notifications targeted at a notification port are always queued on the port, ordered by their associated scheduling priority. The delivery engines belonging to the port service the notification queue, using empty threads from the port pool of which the port is a member. If the port is not a member of a port pool, the delivery engines are blocked.

Port Name Space Semantics

Each distinct right that a port name space contains does not necessarily have a distinct name in the port name space. Receive and send rights to the same port coalesce. For example, 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, receive, port pool, 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_get_refs**.

Although a port name can be explicitly destroyed (**port_destroy**) thereby removing all references, port names are typically manipulated by modifying the user reference count. Calling **port_mod_refs** modifies the reference count for a specified right type associated with a name. 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 when one of its names becomes unusable ("dead") via **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 has been generated. This additional "reference" to the name avoids a race with other threads manipulating the name. A name with a dead-name notification request in effect can be in only one of two 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.

Information about a receive right can be obtained with **port_get_attributes**.

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

Port Pool Semantics

A port pool is created with **port_pool_allocate**, and may contain one or more associated ports. The server populates the pool by moving ports (receive rights only) into it with **port_move** alternatively, a port can be assigned to a port pool when the port is created. Each of the receive rights in the port pool has its own name, as does the pool itself. The server must also assign one or more empty threads (see **Empty Threads**) to the port pool to implement the services it supports.

The state associated with a port pool is:

- a scheduling policy restriction mask
- a queue of threads requesting RPC, exception, or notification service
- a list of empty threads to implement services
- a list of associated ports

Empty threads are associated with a port pool by **thread_create_empty**. We receive only on port pools. Service requests are targeted at a port pool via its associated ports, and a port may be a member of at most one port pool. A thread invoking requests targeted at a port pool with no available empty threads (either because all are currently in use or because none have yet been associated with the port pool) will be blocked. Invoking threads are queued for service by scheduling priority. When an empty thread becomes available, the most eligible blocked invoking thread donates the necessary resources to the empty thread which then executes the requested service.

The scheduling policy mask can be queried and reset using **port_pool_get_attributes** and **port_pool_set_attributes**. By default, a port pool will accept invocations from a thread with any scheduling policy. **port_pool_set_attributes** may be used to set a mask of acceptable scheduling policies; attempts at RPC invocation from threads with other policies will return an error. Delivery engines implicitly created to service notifications simply inherit their scheduling policy from their creating task.

Ports are moved in and out of port pools by **port_move**. When a port is moved into a port pool any blocked invoking threads are requeued on the port pool by scheduling priority. When a port is moved out of a port pool any blocked invoking threads (whether from user RPC or kernel-generated notification) associated with the port are requeued on the port.

Port pools belong to the task in which they are created. When a task is destroyed, its associated port pools are also destroyed. When a port pool is destroyed, any ports belonging to it are implicitly moved out of the port pool. If the port pool is being destroyed as part of task termination, the task's ports will be destroyed when the task's name space is torn down. When a port pool is being explicitly destroyed, empty threads associated with the port pool are destroyed, and formerly empty threads that are currently engaged in RPC will be destroyed when they attempt to return to the (now defunct) port pool. Note that there is no guarantee that these formerly empty threads will return and be destroyed in a timely manner. Any invokers waiting on the port pool get moved to a port-specific queue and will not be awakened until that port is moved into another port pool or destroyed; further invocations on these ports block. The delivery engines associated with notification ports continue to be associated with their notification ports, but further notifications are queued until the port is moved into another port pool with a non-zero number of empty threads. There is no limit associated with notification delivery, since necessary resources are pre-allocated at the time of the request, and debited against the requesting task.

The RPC Kernel Trap Interface

The RPC system has two sets of traps at the kernel interface, one set designed to accept client data and invoke an RPC, the other designed to accept server reply data and return from servicing an RPC invocation.

There are four sorts of data that move across the kernel trap interface: marshalled user data, user buffers (transferred directly w/o marshalling), port send rights, and descriptions of buffers that are to be used to receive the reply data.

There are at least two trap data formats of interest:

- Self-Describing. The data format is completely self-contained so that one can determine how many of each of the four sorts of data are present and where they are entirely from state in memory. The remote (cross node) case of RPC requires this format.
- Implicit. These trap data formats (and there will in general be multiple) are partially or fully implicit in the trap number. In other words, since the precise format and content of the data supplied is implicit in the trap number used, that data does not need to be fully self-describing (e.g., trap #47 moves 4 words of marshalled data in r1-4, and returns two words plus one port, in r1-2 and r3).

Implicit data formats are used to build optimized traps, where the implicit knowledge is present in both the kernel and the application (in the IDL-generated stubs); these formats are inherently machine-dependent, hence we have machine-dependent stubs. The server runtime code should be integrated into the stubs in such a way that registers used by the kernel pass across the runtime into the stubs. For the most optimized use of registers (i.e., more aggressive than the standard C calling sequence on the platform), the stubs and this runtime will have to use assembly language. Implicit data formats will be designed for each platform based on decisions about which cases are important to optimize, and given knowledge of the characteristics of each platform.

Fully self-describing data is necessary for remote invocations. To avoid the need for the kernel to convert between implicit and self-describing formats, the run-time will help maintain location transparency by choosing the appropriate marshalling method and trap. The kernel will export a test of port locality; the run-time may set up the proper stub upon receipt of the send right (a la subcontracts) or choose the marshalling method dynamically. All of this implies that a receive may be in any format; rather than switching on a format identifier, the run-time support will provide dual receive stubs for methods whose signatures are amenable to optimization, using a separate method-ID for each format for a given method.

On the receive side, the run-time must be able to determine the destination (object) and method (method ID) in order to handle the invocation. Ideally, the implicit data formats will be designed so that the dispatch code doesn't disturb any of the registers involved in argument transmission.

RPC invocation is accomplished in the fully self-describing case by calling **mk_rpc** with three parameters: the target port, the address of the send buffer, and the size of the send buffer. The send buffer is laid out as follows:

method ID

number of integers of marshalled user data

number of OOL descriptors

number of ports

number of OOL reply descriptors (the first one is in the reply buffer)

marshalled user data

pointer/length pairs for OOL user buffers

ports

pointer/length pairs for reply buffers

For the machine-dependent, optimized traps for implicit signatures, which transport their data across the user/kernel boundary in registers, the target port and all user data is put into registers by the stubs, using (inline) functions provided by the kernel. If a copyin/out is necessary to move any of the data, the advantage of using registers is reduced.

It is worth noting that the encoding of RPC data is considerably more compact that the IPC message encoding used in Version 1 of MK++, primarily due to the simplification of functionality. Since automatic deallocation of OOL buffers is no longer supported, these OOL buffers can be passed using only a pointer/length pair. Because only send rights may be transferred via RPC, and send rights may only be made (from a receive right) or copied (from a send right), ports may be specified by name only. The V1 semantics of the MACH_MSG_TYPE_MOVE_SEND port disposition is no longer directly supported but is easily achieved by an additional invocation of **port_mod_refs**.

Address Errors During RPC

During the course of RPC invocation and reply, the source and destination addresses for data being passed may be found to be invalid. This is an unusual situation: an RPC receive buffer may have not been initialized properly, or perhaps some thread executing independently in either the client or the server has inadvertantly deallocated memory referenced by an RPC's arguments. There are four cases:

1. Bad client address during RPC invocation
2. Bad server address during RPC invocation
3. Bad server address during RPC reply
4. Bad client address during RPC reply

In all cases, the invocation or reply trap returns an error. In all cases except #2, the error return correctly notifies the offending party of the problem. The second case represents a type of error that should be caught during a server's development. The client thread receives an error regardless of what the service thread does.

Addressing errors associated with the invocation and handling of optimized RPCs may generate exceptions instead of errors. For example, if an RPC uses a register to transfer a faulty address space reference, the problem won't be discovered until a thread executing in user space uses the reference.

Note that it is possible for data to be partially copied before an address error is encountered; consequently, output buffers may or may not have been modified when an RPC returns an addressing error. Port rights are copied after data so that nothing need be undone at the time of an address error.

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 directly invoking RPCs on 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

The kernel interface allows tasks to allocate memory without explicitly specifying the allocated memory's location. For example, the **vm_allocate** and **vm_map** calling signatures include an option that instructs the kernel to choose the location at which to allocate the requested memory.

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

MK++ exports a set of interfaces that allow user space entities to access and manipulate the system's I/O devices. The kernel implements mediated access to the system's devices by exporting a single *device master* port; access to the system's devices is restricted to those entities that hold send rights to the device master port.

An entity acquires access to a particular device by invoking *device_master*, *ledger_name* → **device_open**. When invoking **device_open**, the caller specifies a mode of access. The **device_open** interface returns a port representing the device, this port permits the caller to access the device as specified by the access mode under which the device was opened. The caller may terminate its use of the device by invoking *device* → **device_close**.

Note that each call to **device_open** returns a separate port, each of which represents a distinct kernel device object. At a given time, a particular hardware device may be represented by multiple kernel device objects. For example, a packet-based network device may be represented by several kernel device objects, each of which represents a distinct communication endpoint.

Each device is modeled as a set of records, the meaning and size of which are specific to and defined by the device's driver. The device interface supports synchronous and asynchronous read and write operations: synchronous operations are invoked via the *device* → **device_read** and *device* → **device_write** interfaces; asynchronous operations are invoked via the *device* → **device_read_request** and *device* → **device_write_request** interfaces.

The *device* → **device_read_packet** and *device* → **device_packet_request** interfaces implement synchronous and asynchronous read operations, respectively, on packet-based network devices. Before packets can be read from a particular device, a packet selection filter must be installed on behalf of the device with a call to the **device_set_filter** interface.

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

The kernel also exports the following device-specific functions to support special purpose devices:

- *device* → **device_set_filter** — This call establishes a packet selection filter for all data that appears on the device. The filter is implemented as a small "program" that is executed at interrupt-level on a simple stack

machine. Packets that meet a filter's selection criteria are made available to the protocol server; the server may read packets by invoking **device_read_packet** or **device_packet_request**.

Each open device instance may be associated with one and only one filter at a time, but multiple packet filters may be active if there are multiple kernel objects representing the network device. When multiple filters are active, a given packet may 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-specific 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 limited access to device hardware registers to allow direct user space manipulation of the device or perhaps direct interaction with the kernel device driver, thereby avoiding the overhead that would otherwise be incurred by invoking each I/O operation via the RPC facility. 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* resolu-

tion 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 notifications (**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 notifications 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.

Synchronizers

A synchronizer is a real time primitive used to assist in concurrency control. MK++ offers synchronizers for counting semaphores, mutexes and conditional variables. These synchronizers are created by an MK++ thread and owned by the MK++ thread's associated task. The termination of the owning task results in the destruction of the synchronizer. A synchronizer can be shared by more than the owning task when the owning task exports a send right to another task's thread.

- *task_name* → **semaphore_create** — Create a semaphore.
- *task_name* → **cv_create** — Create a conditional variable.
- *task_name* → **lock_create** — Create a user mutex.
- *semaphore_name*→ **semaphore_signal** — Increment the semaphore count.
- *semaphore_name*→ **semaphore_read_value** — Read the semaphore count value.
- *semaphore_name*→ **semaphore_wait** — Wait for semaphore resource availability.
- *semaphore_name*→ **semaphore_wait_timed** — Wait a period of time for semaphore resource availability.
- *semaphore_name*→ **semaphore_destroy** — Destroy the semaphore.
- *lock_name*→ **lock_acquire** — Acquire the mutex.
- *lock_name*→ **lock_try** — If the mutex isn't locked, then lock it.
- *lock_name*→ **lock_release** — Release the mutex lock.
- *lock_name*→ **lock_destroy** — Destroy the mutex.
- *conditional_variable_name*→ **cv_wait**— Wait for the conditional variable to be signaled.
- *conditional_variable_name*→ **cv_wait_timed**— Wait a period of time for a cv_signal.
- *conditional_variable_name*→ **cv_signal**— Signal a conditional variable.
- *conditional_variable_name*→ **cv_broadcast**— Signal all threads waiting for the conditional variable.
- *conditional_variable_name*→ **cv_destroy**— Destroy the conditional variable.

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 RPCs 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 requests the missing data by invoking an RPC on the range's abstract memory object. 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 RPCs 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 conducts a dialogue with the various memory managers to maintain this cache, filling and flushing this cache as the kernel sees fit. This dialogue is conducted primarily through notifications; this allows memory man-

agers to optimize the concurrency of their operations while the kernel is free to inititiate paging operations without waiting for one or more memory managers to respond.

The kernel sends notifications to a memory manager via the memory object's abstract memory object port; the memory manager invokes RPCs on the kernel via the memory object's (kernel) memory cache control port. (The kernel-initiated notifications are sent via routines labeled as "Server Interfaces" in the *Kernel Interfaces* document; RPCs invoked by the memory managers are labelled as normal RPC-based 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 4 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

Memory object initialization binds the memory manager's abstract memory object (backing store) to a kernel-based memory cache object (memory image). The memory manager provides the kernel with the abstract memory object port over which the kernel is to send paging notifications, and the kernel furnishes the memory manager with the memory cache control port over which the manager is to invoke paging-related RPCs.

Since the kernel communicates via abstract memory object ports according to specially designed protocols, these ports should not be accessible to client entities as they may confuse the memory manager. Indeed, there are invalid notification sequences 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; memory object representative ports implicitly reference 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 an MK++ task instead of the kernel) so they possess their special properties by the manner in which they are used.

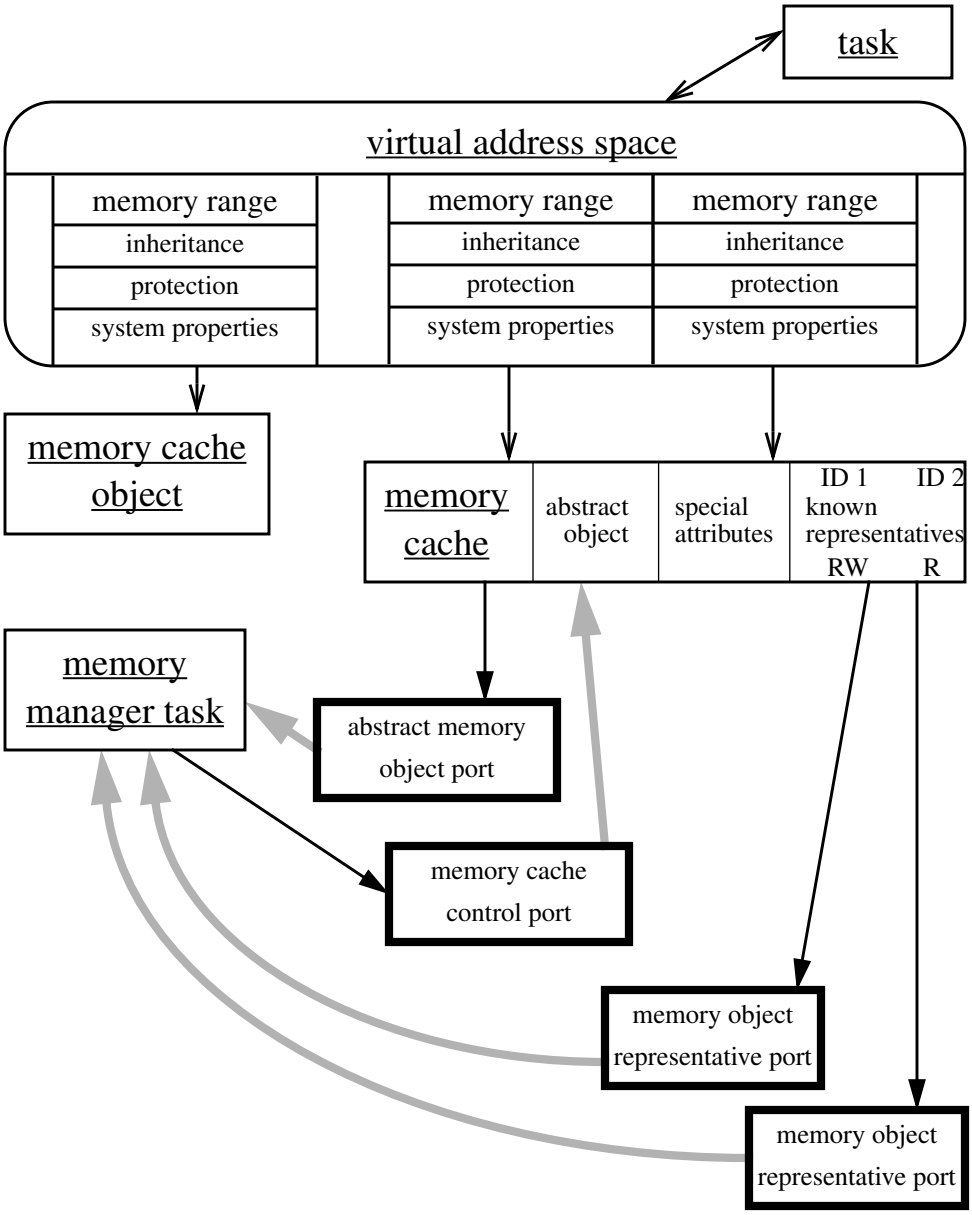


Figure 4 **Virtual Memory Structures**

When a client specifies a memory object representative port to **vm_map** and the kernel does not yet know the identity of the implicit abstract memory object, the kernel engages in a special dialog using the representative port. If the kernel has already been made aware of the abstract memory object, the **vm_map** call can be executed directly with no interaction with the memory manager.

This special dialog is conducted as follows:

1. A task issues *task* → **vm_map** (*memory_object_representative*).

2. *task*'s kernel sends the following notification: *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 a notification to the *abstract_memory_object* port, all of these notifications 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*) notification. 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 notification 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 notification does not give away the cache control port, 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 issue arbitrary RPCs. The supplied port may even be the cache control port for some other memory object.

The memory manager can determine the authenticity of the host name port given to it so it knows when it sends its establish RPC 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 assume that all notifications appearing on the port have come from the kernel.

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

By receiving a notification 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 potential vulnerability for the manager.

The **memory_object_establish** RPC 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 has already been initialized, the memory manager will respond to all subsequent **memory_object_notify** notifications (from the same kernel) by invoking *host_name* (*memory_cache_control*, *memory_object_representative*) → **memory_object_representative**. This RPC 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. Upon receiving this RPC, the kernel destroys the (redundant and useless) memory cache control port.

It is possible, though, for a **memory_object_representative** RPC 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 ignores the **memory_object_representative** RPC because the object—from the kernel's point of view—is no longer initialized.

The manager detects this situation by the presence of a **memory_object_terminate** notification on the abstract memory object port. In this situation, the kernel re-issues the **memory_object_notify** message; the manager should respond once again by invoking **memory_object_establish**.

As a general rule, unless the manager knows for sure that a memory object is not initialized (the object never was initialized or the manager received a **memory_object_terminate** notification), the manager should respond to **memory_object_notify** with **memory_object_representative**. If the kernel was not expecting the **memory_object_representative** RPC, perhaps because a client fabricated the previous **memory_object_notify** RPC), the kernel simply ignores it.

The fact that the manager invoked this RPC means that the manager believes the memory object to be initialized and so the **memory_object_notify** notification 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** RPC 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 by issuing *abstract_memory_object* → **memory_object_data_request** notifications. Typically, the kernel requests (and supplies) a single page per notification. 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 kernel-requested data by invoking the *memory_cache_control* → **memory_object_data_supply** RPC. This RPC 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 invokes *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 pages belonging to a given memory object, the modified pages are sent as out-of-line data via the *abstract_memory_object* → **memory_object_data_return** RPC.

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 notifications for the same pages.

Memory Object Termination

When all tasks remove their mappings to a particular memory object representative, the kernel dissolves the association between the representative and underlying abstract memory object (c.f. object cacheability below) without notifying the memory manager. Consequently, the memory manager must be prepared to receive **memory_object_notify** notifications for a representative at any time. Each subsequent **memory_object_notify** notification 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 notification.

Alternatively, the memory manager may invoke *memory_cache_control* → **memory_object_destroy** to explicitly terminate 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 notification 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** notification). 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.

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** RPC. The memory manager can be informed of the acceptance of these pages by requesting a notification matching the data supply call. This notification, *reply_port* → **memory_object_supply_completed**, identifies which pages were accepted.

The notification will follow all data return messages resulting from rejected supplied pages. The **memory_object_data_return** notification returns modified pages and precious pages (with a flag indicating the case).

Direct Cache Manipulation

Sometimes the memory manager needs to force the flushing of pages, or to alter protections or the like. This is done with the *memory_cache_control* → **memory_object_lock_request** RPC. This call informs the kernel of the operations to be performed. When completed, the kernel issues a *reply_port* → **memory_object_lock_completed** notification.

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 notification to the memory manager via *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 attempts always generate memory access exceptions.) If the memory manager honors an unlock request, it responds by invoking **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** notification to the manager requesting the page with whatever access modes are necessary to satisfy the thread's attempted access. Once a page is in this state, all threads attempting to access the page stall while waiting for it to be supplied. The kernel sends no additional notifications 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 is write-accessible.

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 notifications 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 RPC.

Table 1 Page-in Transitions

initial state	data supply[a]	data unavailable[b]	data error[c]
out	ignore	ignore	ignore
fetch	<$\overline{m}$,a,a,x>[d]	<$\overline{m}$,w,w,$\overline{p}$>[e]	out[f]
<–,–,–,–>	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.
<$\overline{m}$,–,–,$\overline{p}$>	out[b]
<$\overline{m}$,–,–,p>	out[c]
<m,–,–,–>	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,–>

Table 3 Memory Access Transitions

initial state	peek[a]	poke[b]
<$\overline{m}$,w,w,–>	<$\overline{m}$,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.

Table 4 Memory "Lock" Operation Transitions

initial state	null access	read access	write access
out	ignore	ignore	ignore
fetch	ignore	ignore	ignore
<–,–,–,–>	<–,n,n,–>	<–,r,r,–>	<–,w,w,–>

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 5 Memory "Clean" Operation Transitions

initial state[a]	null access	read access	write access
out	ignore	ignore	ignore
fetch	ignore	ignore	ignore

Table 5 Memory "Clean" Operation Transitions

initial state[a]	null access	read access	write access
<$\overline{m}$,–,–,–>	<$\overline{m}$,n,n,–>	<$\overline{m}$,r,r,–>	<$\overline{m}$,w,w,–>
<m,–,–,–>	<$\overline{m}$,n,n,–>	<$\overline{m}$,r,r,–>	<$\overline{m}$,w,w,–>

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
<$\overline{m}$,–,–,$\overline{p}$>	out	out	out
<$\overline{m}$,–,–,p>	out	out	evict
<m,–,–,–>	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 subsystem 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 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 **port_allocate**) is the source from which the port is, by default, drawn. 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.

Ledgers are 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 5 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.

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.

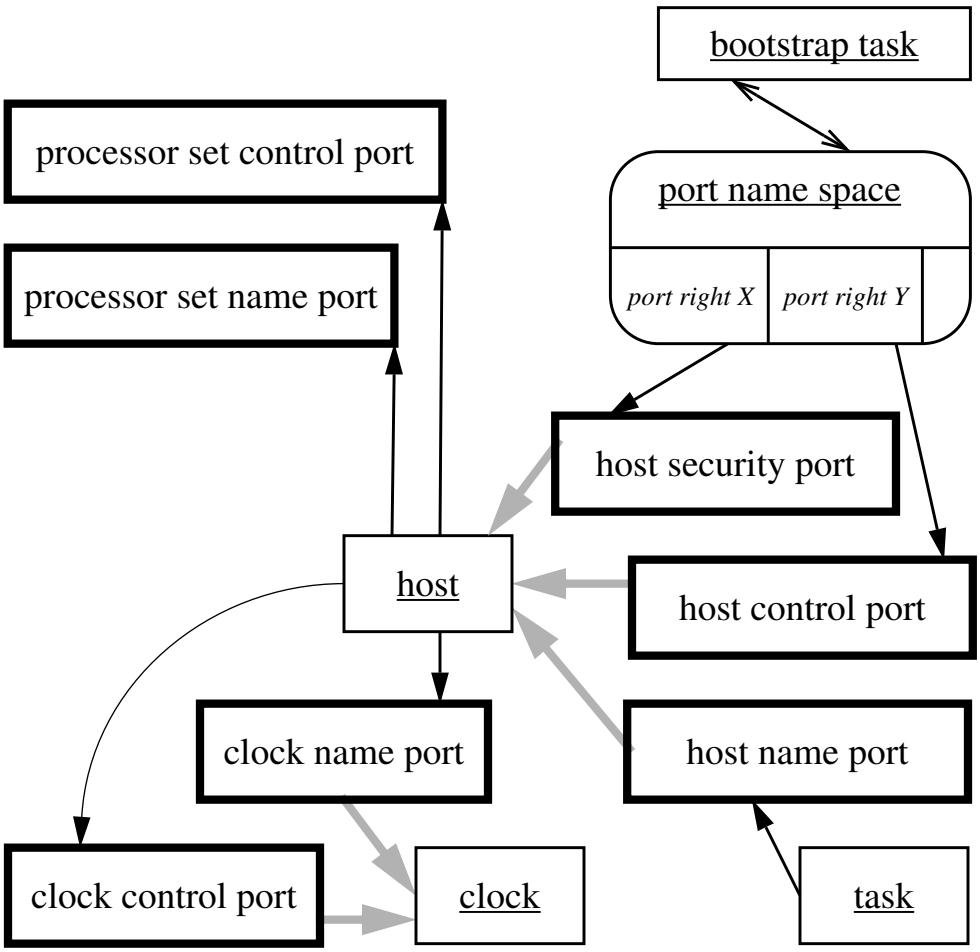


Figure 5 **Host (and Clock) Structures**

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 RPCs invoked by a task's threads are tagged by the kernel with this unforgeable security ID so that servers can reliably determine the identity of the entity that has invoked a given RPC.

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 kernel-generated notifications are tagged with 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 6 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 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.

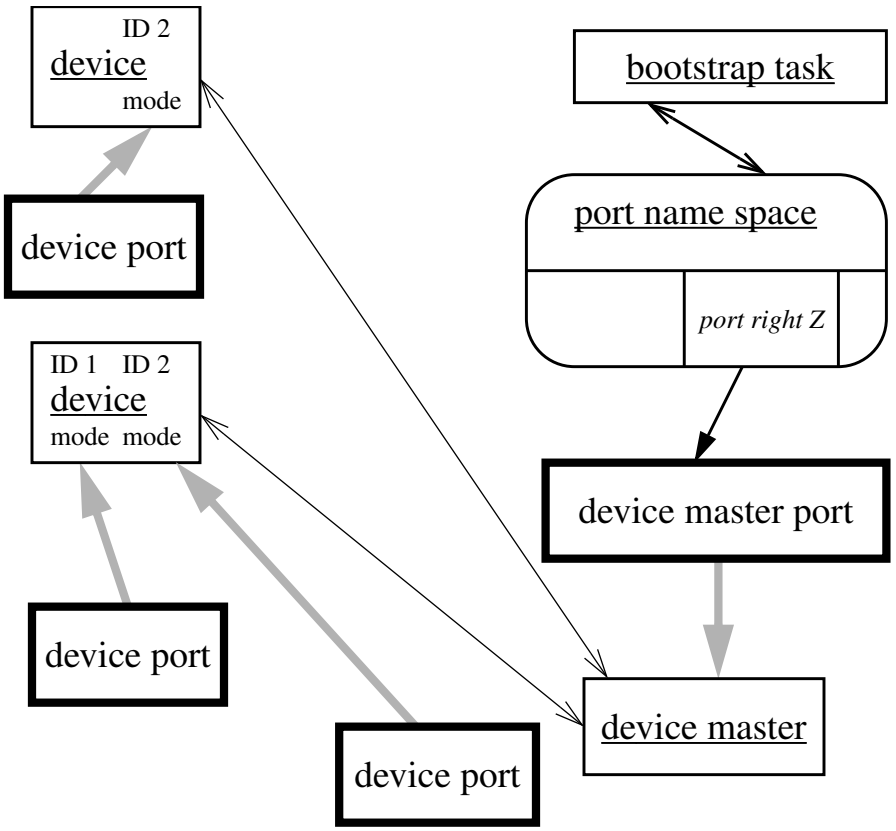


Figure 6 **Kernel Device Structures**

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 invoke and RPC on 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

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 identification tag to each RPC sent from one task to another.
- The kernel applies the kernel's message tag to all notifications 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.