MACH 3 Kernel Interface

Open Software Foundation and Carnegie Mellon University

Keith Loepere, Editor

OSF

May 6, 1991

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

Books in the OSF Mach 3 series:

MACH 3 Kernel Principles

MACH 3 Kernel Interface

Server Writer's Guide

Server Writer's Interface

Revision History:

Revision 0	September 17, 1990	Initial Published Draft
Revision 1	May 6, 1991	Consistent with Kernel Principles document

Change bars indicate changes from revision 0.

This document is partially derived from earlier Mach documents written by Robert V. Baron, Joseph S. Barrera, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, Alessandro Forin, David B. Golub, Richard F. Rashid, Mary R. Thompson, Avadis Tevanian, Jr. and Michael W. Young.

Contents

CHAPTER 1

Introduction

This book documents the various interfaces to the MACH 3 kernel. The text describes each interface to the kernel in isolation. The relationship of interfaces to one another, and the way that interfaces are combined to write user servers is the subject of a companion volume.

The organization of this book is such that it follows the organization of the kernel into its major functional areas. Although the kernel interface is itself not object oriented, the division of interfaces into areas is largely done according to the significant object utilized or manipulated by the interfaces. Each such object receives its own chapter. Of course, the assignment of interfaces into these chapters is a difficult and highly subjective process. For example, an interface that returns the list of processor sets defined for a host can be grouped with host related interfaces or processor set related interfaces. Each interface, though, appears only once in this book.

Appendices give a description of the structures and fields used by these interfaces, a list of possible error return values from the kernel and an alphabetical index of functions and data structures.

Interface Descriptions

Each interface is listed separately, each starting on its own page. For each interface, some or all of the following features are presented:

- the name of the interface
- a brief description
- the pertinent library. For functions, this is the library that contains it and the header file that provides the function prototype. For data structures, this is the header file

that defines it. If two such header files are listed, the first is the preferred file to use, and the file named in parenthesis is the actual file that declares the entity.

- a synopsis of the interface, in C form
- an extended description of the function performed by the call
- a description of each parameter to the call
- additional notes on the use of the interface
- cautions relating to the interface use
- an explanation of the significant return values
- references to related interfaces

General Notes

Most of the interfaces in this book are MIG generated interfaces. That is, they are stub routines generated from MIG interface description files. Calling these interfaces will actually result in a MACH IPC message being sent to the port that is the first argument in the call. This has two important effects.

- These calls may fail for various MIG or IPC related reasons. The list of error returns for these calls should always be considered to also include the IPC related errors (MACH_MSG_..., MACH_SEND_... and MACH_RCV_...) and the MIG related errors (MIG_...).
- These calls only invoke their expected effect when the acting port is indeed a port of the specified type. That is, if a call expects a port that names a task (a kernel task port) and the port is instead a port managed by a task, the routine will still happily generate the appropriate MACH message and send it to that task. What the target task will do with the message is up to it. Note that it is this effect that allows the Net message server to work.

A few of these interfaces are actually system calls. In general, the system calls (with the obvious exception of the **mach_msg** call), work only on the current task or thread. (Some functions are a hybrid; they first try the system call, and, failing that, they try sending a MACH message. This is an optimization for some interfaces for which the target is usually the invoking task or thread.) Any routine not documented as a system call is a MIG stub routine.

Most of these interfaces are of the type **Function**. This means that there is actually a C callable function (most likely in **libmach.a**) that has the calling sequence listed and that when called invokes some kernel or kernel related service. Some interfaces have the type **Server Interface**. Such a description applies to interfaces that are called in server tasks on behalf of messages sent from the kernel. That is, it is assumed that some task is listening (probably with **mach_msg_server**) on a port to which the kernel is to send messages. A received message will be passed to a MIG generated server routine (*service*_**server**) which will call an appropriate server target function. It is these server target functions, one for each different message that the kernel generates, that are listed as **Server Interface**s.

CHAPTER 2 IPC Interface Specifications

This chapter discusses the specifics of the kernel's inter-"process" communication (IPC) interfaces. The interfaces discussed are only the interfaces directly involved in sending and receiving IPC messages.

mach_msg

mach_msg Function — Sends and receives a message using the same message buffer

LIBRARY

libmach.a

#include <mach.h> (<mach/message.h>)

SYNOPSIS

mach_msg_return_t **mach_msg** (mach_msg_header_t **msg,*
mach_msg_option_t *option,* mach_msg_size_t
send_size, mach_msg_size_t *rcv_size,* mach_port_t
rcv_name, mach_msg_timeout_t *timeout,*
mach_port_t *notify)*;

DESCRIPTION

The **mach_msg** system call sends and receives Mach messages. Mach messages contain typed data, which can include port rights and addresses of large regions of memory.

If the *option* argument contains MACH_SEND_MSG, it sends a message. The *send_size* argument specifies the size of the message to send. The *msgh_-remote_port* field of the message header specifies the destination of the message.

If the *option* argument contains MACH_RCV_MSG, it receives a message. The *rcv_size* argument specifies the size of the message buffer that will receive the message; messages larger than *rcv_size* are not received. The *rcv_name* argument specifies the port or port set from which to receive.

If the *option* argument contains both MACH_SEND_MSG and MACH_-RCV_MSG, then **mach_msg** does both send and receive operations. If the send operation encounters an error (any return code other than MACH_MSG_SUCCESS), then the call returns immediately without attempting the receive operation. Semantically the combined call is equivalent to separate send and receive calls, but it saves a system call and enables other internal optimizations.

If the *option* argument specifies neither MACH_SEND_MSG nor MACH_-RCV_MSG, then **mach_msg** does nothing.

Some options, like MACH_SEND_TIMEOUT and MACH_RCV_TIMEOUT, share a supporting argument. If these options are used together, they make independent use of the supporting argument's value.

PARAMETERS

msg
: [in/out] A message buffer. This should be aligned on a long-word boundary.

option
: Message options are bit values, combined with bitwise-or. One or both of MACH_SEND_MSG and MACH_RCV_MSG should be used.Other options act as modifiers.

send_size
: When sending a message, specifies the size of the message buffer. Otherwise zero should be supplied.

rcv_size
: When receiving a message, specifies the size of the message buffer. Otherwise zero should be supplied.

rcv_name
: When receiving a message, specifies the port or port set. Otherwise MACH_PORT_NULL should be supplied.

timeout
: When using the MACH_SEND_TIMEOUT and MACH_RCV_TIMEOUT options, specifies the time in milliseconds to wait before giving up. Otherwise MACH_MSG_TIMEOUT_NONE should be supplied.

notify
: When using the MACH_SEND_NOTIFY, MACH_SEND_CANCEL, and MACH_RCV_NOTIFY options, specifies the port used for the notification. Otherwise MACH_PORT_NULL should be supplied.

NOTES

The Mach kernel provides message-oriented, capability-based inter-process communication. The inter-process communication (IPC) primitives efficiently support many different styles of interaction, including remote procedure calls, object-oriented distributed programming, streaming of data, and sending very large amounts of data.

Major Concepts

The IPC primitives operate on three abstractions: messages, ports, and port sets. User tasks access all other kernel services and abstractions via the IPC primitives.

The message primitives let tasks send and receive messages. Tasks send messages to ports. Messages sent to a port are delivered reliably (messages may not be lost) and are received in the order in which they were sent. Messages contain a fixed-size header and a variable amount of typed data following the header. The header describes the destination and size of the message.

The IPC implementation makes use of the VM system to efficiently transfer large amounts of data. The message body can contain an address of a region of the sender's address space which should be transferred as part of the message. When a task receives a message containing an out-of-line region of data, the data appears in an unused portion of the receiver's address space. This transmission of out-of-line data is optimized so that sender and receiver share the physical pages of data copy-on-write, and no actual data copy occurs unless the pages are written. Regions of memory up to the size of a full address space may be sent in this manner.

Ports hold a queue of messages. Tasks operate on a port to send and receive messages by exercising capabilities (rights) for the port. Multiple tasks can hold send rights for a port. Tasks can also hold send-once rights, which grant the ability to send a single message. Only one task can hold the receive capability (receive right) for a port. Port rights can be transferred between tasks via messages. The sender of a message can specify in the message body that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.

Tasks can receive messages from ports and port sets. The port set abstraction allows a single thread to wait for a message from any of several ports. Tasks manipulate port sets with a port set name, which is taken from the same name space as are the port rights. The port-set name may not be transferred in a message. A port set holds receive rights, and a receive operation on a port set blocks waiting for a message sent to any of the constituent ports. A port may not belong to more than one port set, and if a port is a member of a port set, the holder of the receive right can't receive directly from the port.

Port rights are a secure, location-independent way of naming ports. The port queue is a protected data structure, only accessible via the kernel's exported message primitives. Rights are also protected by the kernel; there is no way for a malicious user task to guess a port's internal name and send a message to a port to which it shouldn't have access. Port rights do not carry any location information. When a receive right for a port moves from task to task, and even between tasks on different machines, the send rights for the port remain unchanged and continue to function.

Port Rights

Each task has its own space of port rights. Port rights are named with positive integers. Except for the reserved values MACH_PORT_NULL (0) and MACH_PORT_DEAD (-1), this is a full 32-bit name space. When the kernel chooses a name for a new right, it is free to pick any unused name (one which denotes no right) in the space.

There are three basic kinds of rights: receive rights, send rights and send-once rights. A port name can name any of these types of rights, a port-set, be a dead name, or name nothing. Dead names are not capabilities. They act as place-holders to prevent a name from being otherwise used.

A port is destroyed, or dies, when its receive right is de-allocated. When a port dies, send and send-once rights for the port turn into dead names. Any messages queued at the port are destroyed, which de-allocates the port rights and out-of-line memory in the messages.

Tasks may hold multiple user-references for send rights and dead names. When a task receives a send right which it already holds, the kernel increments the right's user-reference count. When a task de-allocates a send right, the kernel decrements its user-reference count, and the task only loses the send right when the count goes to zero.

Send-once rights always have a user-reference count of one, although a port can have multiple send-once rights, because each send-once right held by a task has a different name. In contrast, when a task holds send rights or a receive right for a port, the rights share a single name.

A message body can carry port rights; the *msgt_name* (*msgtl_name*) field in a type descriptor specifies the type of port right and how the port right is to be extracted from the caller. The values MACH_PORT_NULL and MACH_PORT_DEAD are always valid in place of a port right in a message body.

In a sent message, the following *msgt_name* values denote port rights:

MACH_MSG_TYPE_MAKE_SEND
: The message will carry a send right, but the caller must supply a receive right. The send right is created from the receive right, and the receive right's make-send count is incremented.

MACH_MSG_TYPE_COPY_SEND
: The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is not changed. The caller may also supply a dead name and the receiving task will get MACH_PORT_DEAD.

MACH_MSG_TYPE_MOVE_SEND
: The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is decremented, and the right is destroyed if the count becomes zero. Unless a receive right remains, the name becomes available for recycling. The caller may also supply a dead name, which loses a user reference, and the receiving task will get MACH_PORT_DEAD.

MACH_MSG_TYPE_MAKE_SEND_ONCE
: The message will carry a send-once right, but the caller must supply a receive right. The send-once right is created from the receive right.

MACH_MSG_TYPE_MOVE_SEND_ONCE
: The message will carry a send-once right, and the caller should supply a send-once right. The caller loses the supplied send-once right.

The caller may also supply a dead name, which loses a user reference, and the receiving task will get MACH_PORT_DEAD.

MACH_MSG_TYPE_MOVE_RECEIVE
: The message will carry a receive right, and the caller should supply a receive right. The caller loses the supplied receive right, but retains any send rights with the same name.

If a message carries a send or send-once right, and the port dies while the message is in transit, then the receiving task will get MACH_PORT_DEAD instead of a right.

The following *msgt_name* values in a received message indicate that it carries port rights:

MACH_MSG_TYPE_PORT_SEND
: This value is an alias for MACH_MSG_TYPE_MOVE_SEND. The message carried a send right. If the receiving task already has send and/or receive rights for the port, then that name for the port will be reused. Otherwise, the new right will have a new, previously unused, name. If the task already has send rights, it gains a user reference for the right (unless this would cause the user-reference count to overflow). Otherwise, it acquires send rights, with a user-reference count of one.

MACH_MSG_TYPE_PORT_SEND_ONCE
: This value is an alias for MACH_MSG_TYPE_MOVE_SEND_ONCE. The message carried a send-once right. The right will have a new, previously unused, name.

MACH_MSG_TYPE_PORT_RECEIVE
: This value is an alias for MACH_MSG_TYPE_MOVE_RECEIVE. The message carried a receive right. If the receiving task already has send rights for the port, then that name for the port will be reused. Otherwise, the right will have a new, previously unused, name. The make-send count of the receive right is reset to zero, but the port retains other attributes like queued messages, extant send and send-once rights, and requests for port-destroyed and no-senders notifications. (Note: It is currently planned to remove port-destroyed notifications from the kernel interface and to define no-senders notifications as being canceled when a receive right is moved.)

Memory

A message body can contain an address of a region of the sender's address space which should be transferred as part of the message. The message carries a logical copy of the memory, but the kernel uses VM techniques to defer any actual page copies. Unless the sender or the receiver modifies the data, the physical pages remain shared.

An out-of-line transfer occurs when the data's type descriptor specifies *msgt_inline* as FALSE. The address of the memory region should follow the type descriptor in the message body. The type descriptor and the address contribute to the message's size (*send_size*, *msgh_size*). The out-of-line data does not contribute to the message's size.

The name, size, and number fields in the type descriptor describe the type and length of the out-of-line data, not the address. Out-of-line memory frequently requires long type descriptors (**mach_msg_type_long_t**), because the *msgt_-number* field is too small to describe a page of 4K bytes.

Out-of-line memory arrives somewhere in the receiver's address space as new memory. It has the same inheritance and protection attributes as newly **vm_allocate**'ed memory. The receiver has the responsibility of de-allocating (with **vm_deallocate**) the memory when it is no longer needed. Security-conscious receivers should exercise caution when dealing with out-of-line memory from un-trustworthy sources, because the memory may be backed by an unreliable memory manager.

Null out-of-line memory is legal. If the out-of-line region size is zero (for example, because *msgtl_number* is zero), then the region's specified address is ignored. A received null out-of-line memory region always has a zero address.

Unaligned addresses and region sizes that are not page multiples are legal. A received message can also contain regions with unaligned addresses and funny sizes. In the general case, the first and last pages in the new memory region in the receiver do not contain data from the sender, but are partly zero. The received address points into the middle of the first page. This possibility doesn't complicate de-allocation, because **vm_deallocate** does the right thing, rounding the start address down and the end address up to de-allocate all arrived pages.

Out-of-line memory has a de-allocate option, controlled by the *msgt_deallocate* bit. If it is TRUE and the out-of-line memory region is not null, then the region is implicitly de-allocated from the sender, as if by **vm_deallocate**. In particular, the start and end addresses are rounded so that every page overlapped by the memory region is de-allocated. The use of *msgt_deallocate* effectively changes the memory copy into a memory movement. In a received message, *msgt_deallocate* is TRUE in type descriptors for out-of-line memory.

Out-of-line memory can carry port rights.

Message Send

The send operation queues a message to a port. The message carries a copy of the caller's data. After the send, the caller can freely modify the message buffer or the out-of-line memory regions and the message contents will remain unchanged.

Message delivery is reliable and sequenced. Messages are not lost, and messages sent to a port from a single thread are received in the order in which they were sent.

If the destination port's queue is full, then several things can happen. If the message is sent to a send-once right (*msgh_remote_port* carries a send-once right), then the kernel ignores the queue limit and delivers the message. Otherwise the caller blocks until there is room in the queue, unless the MACH_SEND_TIMEOUT or MACH_SEND_NOTIFY options are used. If a port has several blocked senders, then any of them may queue the next message when space in the queue becomes available, with the proviso that a blocked sender will not be indefinitely starved.

These options modify MACH_SEND_MSG. If MACH_SEND_MSG is not also specified, they are ignored.

MACH_SEND_TIMEOUT
: The *timeout* argument should specify a maximum time (in milliseconds) for the call to block before giving up. If the message can't be queued before the timeout interval elapses, then the call returns MACH_SEND_TIMED_OUT. A zero timeout is legitimate.

MACH_SEND_NOTIFY
: The *notify* argument should specify a receive right for a notify port. If the send were to block, then instead the message is queued, MACH_SEND_WILL_NOTIFY is returned, and a msg-accepted notification is requested. If MACH_SEND_TIMEOUT is also specified, then MACH_SEND_NOTIFY doesn't take effect until the timeout interval elapses.

  Only one message at a time can be forcibly queued to a send right with MACH_SEND_NOTIFY. A msg-accepted notification is sent to the notify port when another message can be forcibly queued. If an attempt is made to use MACH_SEND_NOTIFY before then, the call returns a MACH_SEND_NOTIFY_IN_PROGRESS error.

  The msg-accepted notification carries the name of the send right. If the send right is de-allocated before the msg-accepted notification is generated, then the msg-accepted notification carries the value MACH_PORT_NULL. If the destination port is destroyed before the notification is generated, then a send-once notification is generated instead.

  (Note: It is currently planned that this option will be deleted, as well as the provision of the corresponding notification.)

MACH_SEND_INTERRUPT
: If specified, the **mach_msg** call will return MACH_SEND_INTERRUPTED if a software interrupt aborts the call. Otherwise, the send operation will be retried.

MACH_SEND_CANCEL
: The *notify* argument should specify a receive right for a notify port. If the send operation removes the destination port right from the caller, and the removed right had a dead-name request registered for it, and *notify* is the notify port for the dead-name request, then the dead-

name request may be silently canceled (instead of resulting in what would have been a port-deleted notification).

This option is typically used to cancel a dead-name request made with the MACH_RCV_NOTIFY option. It should only be used as an optimization.

Some return codes, like MACH_SEND_TIMED_OUT, imply that the message was almost sent, but could not be queued. In these situations, the kernel tries to return the message contents to the caller with a pseudo-receive operation. This prevents the loss of port rights or memory which only exist in the message. For example, a receive right which was moved into the message, or out-of-line memory sent with the de-allocate bit.

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

The pseudo-receive operation may encounter resource shortages. This is similar to a MACH_RCV_BODY_ERROR return code from a receive operation. When this happens, the normal send return codes are augmented with the MACH_MSG_IPC_SPACE, MACH_MSG_VM_SPACE, MACH_MSG_IPC_KERNEL, and MACH_MSG_VM_KERNEL bits to indicate the nature of the resource shortage.

The queueing of a message carrying receive rights may create a circular loop of receive rights and messages, which can never be received. For example, a message carrying a receive right can be sent to that receive right. This situation is not an error, but the kernel will garbage-collect such loops, destroying the messages.

Message Receive

The receive operation de-queues a message from a port. The receiving task acquires the port rights and out-of-line memory regions carried in the message.

The *rcv_name* argument specifies a port or port set from which to receive. If a port is specified, the caller must possess the receive right for the port and the port must not be a member of a port set. If no message is present, then the call blocks, subject to the MACH_RCV_TIMEOUT option.

If a port set is specified, the call will receive a message sent to any of the member ports. It is permissible for the port set to have no member ports, and ports may be added and removed while a receive from the port set is in progress. The received message can come from any of the member ports which have messages, with the proviso that a member port with messages will not be indefinitely starved. The *msgh_local_port* field in the received message header specifies from which port in the port set the message came.

The *rcv_size* argument specifies the size of the caller's message buffer. The **mach_msg** call will not receive a message larger than *rcv_size*. Messages that are too large are destroyed, unless the MACH_RCV_LARGE option is used.

The destination and reply ports are reversed in a received message header. The *msgh_local_port* field carries the name of the destination port, from which the message was received, and the *msgh_remote_port* field carries the reply port right. The bits in *msgh_bits* are also reversed. The MACH_MSGH_BITS_LOCAL bits have the value MACH_MSG_TYPE_PORT_SEND if the message was sent to a send right, and the value MACH_MSG_TYPE_PORT_SEND_ONCE if was sent to a send-once right. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

A received message can contain port rights and out-of-line memory. The *msgh_local_port* field does not carry a port right; the act of receiving the message destroys the send or send-once right for the destination port. The *msgh_remote_port* field does carry a port right, and the message body can carry port rights and memory if MACH_MSGH_BITS_COMPLEX is present in *msgh_bits*. Received port rights and memory should be consumed or de-allocated in some fashion.

In almost all cases, *msgh_local_port* will specify the name of a receive right, either *rcv_name*, or, if *rcv_name* is a port set, a member of *rcv_name*. If other threads are concurrently manipulating the receive right, the situation is more complicated. If the receive right is renamed during the call, then *msgh_local_port* specifies the right's new name. If the caller loses the receive right after the message was de-queued from it, then **mach_msg** will proceed instead of returning MACH_RCV_PORT_DIED. If the receive right was destroyed, then *msgh_local_port* specifies MACH_PORT_DEAD. If the receive right still exists, but isn't held by the caller, then *msgh_local_port* specifies MACH_PORT_NULL.

These options modify MACH_RCV_MSG. If MACH_RCV_MSG is not also specified, they are ignored.

MACH_RCV_TIMEOUT
: The *timeout* argument should specify a maximum time (in milliseconds) for the call to block before giving up. If no message arrives before the timeout interval elapses, then the call returns MACH_RCV_TIMED_OUT. A zero timeout is legitimate.

MACH_RCV_NOTIFY
: The *notify* argument should specify a receive right for a notify port. If receiving the reply port creates a new port right in the caller, then the notify port is used to request a dead-name notification for the new port right.

MACH_RCV_INTERRUPT
: If specified, the **mach_msg** call will return MACH_RCV_INTERRUPTED if a software interrupt aborts the call. Otherwise, the receive operation will be retried.

MACH_RCV_LARGE
: If the message is larger than *rcv_size*, then the message remains queued instead of being destroyed. The call returns MACH_RCV_TOO_LARGE and the actual size of the message is returned in the *msgh_size* field of the message header.

If a resource shortage prevents the reception of a port right, the port right is destroyed and the caller sees the name MACH_PORT_NULL. If a resource shortage prevents the reception of an out-of-line memory region, the region is destroyed and the caller sees a zero address. In addition, the *msgt_size* (*msgtl_size*) field in the region's type descriptor is changed to zero. If a resource shortage prevents the reception of out-of-line memory carrying port rights, then the port rights are always destroyed if the memory region can not be received. A task never receives port rights or memory for which it is not told.

The MACH_RCV_HEADER_ERROR return code indicates a resource shortage in the reception of the message's header. The reply port and all port rights and memory in the message body are destroyed. The caller receives the message's header, with all fields correct except for the reply port.

The MACH_RCV_BODY_ERROR return code indicates a resource shortage in the reception of the message's body. The message header, including the reply port, is correct. The kernel attempts to transfer all port rights and memory regions in the body, and only destroys those that can't be transferred.

Atomicity

The **mach_msg** call handles port rights in a message header atomically. Port rights and out-of-line memory in a message body do not enjoy this atomicity guarantee. The message body may be processed front-to-back, back-to-front, first out-of-line memory then port rights, in some random order, or even atomically.

For example, consider sending a message with the destination port specified as MACH_MSG_TYPE_MOVE_SEND and the reply port specified as MACH_MSG_TYPE_COPY_SEND. The same send right, with one user-reference, is supplied for both the *msgh_remote_port* and *msgh_local_port* fields. Because **mach_msg** processes the message header atomically, this succeeds. If *msgh_remote_port* were processed before *msgh_local_port*, then **mach_msg** would return MACH_SEND_INVALID_REPLY in this situation.

On the other hand, suppose the destination and reply port are both specified as MACH_MSG_TYPE_MOVE_SEND, and again the same send right with one user-reference is supplied for both. Now the send operation fails, but because it processes the header atomically, **mach_msg** can return either MACH_SEND_INVALID_DEST or MACH_SEND_INVALID_REPLY.

For example, consider receiving a message at the same time another thread is de-allocating the destination receive right. Suppose the reply port field carries a send right for the destination port. If the de-allocation happens before the dequeuing, then the receiver gets MACH_RCV_PORT_DIED. If the de-allocation happens after the receive, then the *msgh_local_port* and the *msgh_remote_port* fields both specify the same right, which becomes a dead name when the receive right is de-allocated. If the de-allocation happens between the de-queue and the receive, then the *msgh_local_port* and *msgh_remote_port* fields both specify MACH_PORT_DEAD. Because the header is processed atomically, it is not possible for just one of the two fields to hold MACH_PORT_DEAD.

The MACH_RCV_NOTIFY option provides a more likely example. Suppose a message carrying a send-once right reply port is received with MACH_RCV_NOTIFY at the same time the reply port is destroyed. If the reply port is destroyed first, then *msgh_remote_port* specifies MACH_PORT_DEAD and the kernel does not generate a dead-name notification. If the reply port is destroyed after it is received, then *msgh_remote_port* specifies a dead name for which the kernel generates a dead-name notification. It is not possible to receive the reply port right and have it turn into a dead name before the dead-name notification is requested; as part of the message header the reply port is received atomically.

Implementation

mach_msg is a wrapper for a system call. **mach_msg** has the responsibility for repeating the interrupted system call.

CAUTIONS

Sending out-of-line memory with a non-page-aligned address, or a size which is not a page multiple, works but with a caveat. The extra bytes in the first and last page of the received memory are not zeroed, so the receiver can peek at more data than the sender intended to transfer. This might be a security problem for the sender.

If MACH_RCV_TIMEOUT is used without MACH_RCV_INTERRUPT, then the timeout duration might not be accurate. When the call is interrupted and automatically retried, the original timeout is used. If interrupts occur frequently enough, the timeout interval might never expire. MACH_SEND_TIMEOUT without MACH_SEND_INTERRUPT suffers from the same problem.

RETURN VALUE

The send operation can generate the following return codes. These return codes imply that the call did nothing:

MACH_SEND_MSG_TOO_SMALL
: The specified *send_size* was smaller than the minimum size for a message.

MACH_SEND_NO_BUFFER
: A resource shortage prevented the kernel from allocating a message buffer.

MACH_SEND_INVALID_DATA
: The supplied message buffer was not readable.

MACH_SEND_INVALID_HEADER
: The *msgh_bits* value was invalid.

MACH_SEND_INVALID_DEST
: The *msgh_remote_port* value was invalid.

MACH_SEND_INVALID_REPLY
: The *msgh_local_port* value was invalid.

MACH_SEND_INVALID_NOTIFY
: When using MACH_SEND_CANCEL, the *notify* argument did not denote a valid receive right.

These return codes imply that some or all of the message was destroyed:

MACH_SEND_INVALID_MEMORY
: The message body specified out-of-line data that was not readable.

MACH_SEND_INVALID_RIGHT
: The message body specified a port right which the caller didn't possess.

MACH_SEND_INVALID_TYPE
: A type descriptor was invalid.

MACH_SEND_MSG_TOO_SMALL
: The last data item in the message ran over the end of the message.

These return codes imply that the message was returned to the caller with a pseudo-receive operation:

MACH_SEND_TIMED_OUT
: The *timeout* interval expired.

MACH_SEND_INTERRUPTED
: A software interrupt occurred.

MACH_SEND_INVALID_NOTIFY
: When using MACH_SEND_NOTIFY, the *notify* argument did not denote a valid receive right.

MACH_SEND_NO_NOTIFY
: A resource shortage prevented the kernel from setting up a msg-accepted notification.

MACH_SEND_NOTIFY_IN_PROGRESS
: A msg-accepted notification was already requested, and hasn't yet been generated.

These return codes imply that the message was queued:

MACH_SEND_WILL_NOTIFY
: The message was forcibly queued, and a msg-accepted notification was requested.

MACH_MSG_SUCCESS
: The message was queued.

The receive operation can generate the following return codes. These return codes imply that the call did not de-queue a message:

MACH_RCV_INVALID_NAME
: The specified *rcv_name* was invalid.

MACH_RCV_IN_SET
: The specified port was a member of a port set.

MACH_RCV_TIMED_OUT
: The *timeout* interval expired.

MACH_RCV_INTERRUPTED
: A software interrupt occurred.

MACH_RCV_PORT_DIED
: The caller lost the rights specified by *rcv_name*.

MACH_RCV_PORT_CHANGED
: *rcv_name* specified a receive right which was moved into a port set during the call.

MACH_RCV_TOO_LARGE
: When using MACH_RCV_LARGE, and the message was larger than *rcv_size*. The message is left queued, and its actual size is returned in the *msgh_size* field of the message buffer.

These return codes imply that a message was de-queued and destroyed:

MACH_RCV_HEADER_ERROR
: A resource shortage prevented the reception of the port rights in the message header.

MACH_RCV_INVALID_NOTIFY
: When using MACH_RCV_NOTIFY, the *notify* argument did not denote a valid receive right.

MACH_RCV_TOO_LARGE
: When not using MACH_RCV_LARGE, a message larger than *rcv_size* was de-queued and destroyed.

These return codes imply that a message was received:

MACH_RCV_BODY_ERROR
: A resource shortage prevented the reception of a port right or out-of-line memory region in the message body.

MACH_RCV_INVALID_DATA
: The specified message buffer was not writable. The calling task did successfully receive the port rights and out-of-line memory regions in the message.

MACH_MSG_SUCCESS
A message was received.

Resource shortages can occur after a message is de-queued, while transferring port rights and out-of-line memory regions to the receiving task. The **mach_msg** call returns MACH_RCV_HEADER_ERROR or MACH_RCV_-BODY_ERROR in this situation. These return codes always carry extra bits (bitwise-or'ed) that indicate the nature of the resource shortage:

MACH_MSG_IPC_SPACE
There was no room in the task's IPC name space for another port name.

MACH_MSG_VM_SPACE
There was no room in the task's VM address space for an out-of-line memory region.

MACH_MSG_IPC_KERNEL
A kernel resource shortage prevented the reception of a port right.

MACH_MSG_VM_KERNEL
A kernel resource shortage prevented the reception of an out-of-line memory region.

RELATED INFORMATION

Functions: **mach_msg_receive**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_-type_long, mach_msg_accepted_notification**, **mach_send_once_notification**.

mach_msg_receive

mach_msg_receive Function — Receives a message from a port or port set

LIBRARY

libmach.a

SYNOPSIS

msg_return_t **mach_msg_receive**(mach_msg_header_t **header);*

DESCRIPTION

The **mach_msg_receive** function is a shorthand for the following call:

mach_msg (*header*, MACH_RCV_MSG, 0, *header*->*msgh_size*, *header*->*msgh_local_port*,
MACH_MSG_TIMEOUT_NONE,
MACH_PORT_NULL);

PARAMETERS

header
: [in/out] The address of the buffer that is to receive the message. The *msgh_local_port* and *msgh_size* fields in *header* must be set.

RETURN VALUE

Refer to **mach_msg** for a description of the various receive errors.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_send**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_type_long**.

mach_msg_send

mach_msg_send Function — Sends a message to a port

LIBRARY

libmach.a

SYNOPSIS

msg_return_t **mach_msg_send**(mach_msg_header_t **header*);

DESCRIPTION

The **mach_msg_send** function is a shorthand for the following call:

mach_msg (*header*, MACH_SEND_MSG, *header*->*msgh_size*, 0,
MACH_PORT_NULL,
MACH_MSG_TIMEOUT_NONE,
MACH_PORT_NULL);

PARAMETERS

header
The address of the buffer that contains the message to be sent.

RETURN VALUE

Refer to **mach_msg** for a description of the send errors.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_receive**.

Data Structures: **mach_msg_header**, **mach_msg_type**, **mach_msg_type_long**.

CHAPTER 3

Port Manipulation Interface Specifications

This chapter discusses the specifics of the kernel's port manipulation interfaces. This includes port, port set and port right related functions. Also included are interfaces that return port related status information that applies to a single task.

do_mach_notify_dead_name

do_mach_notify_dead_name Server Interface — Handles the occurrence of a dead-name notification

SYNOPSIS

kern_return_t **do_mach_notify_dead_name** (notify_port_t *notify,*
mach_port_name_t *name)*;

DESCRIPTION

A **do_mach_notify_dead_name** function is called as the result of a kernel message indicating that a port's receiver is gone, thereby turning the right into a dead name. *notify* is the port named via **mach_port_request_notification**.

PARAMETERS

notify
The port to which the notification was sent.

name
The dead name.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_msg_accepted

do_mach_notify_msg_accepted Server Interface — Handles the occurrence of a message accepted notification

SYNOPSIS

kern_return_t **do_mach_notify_msg_accepted** (notify_port_t *notify,*
mach_port_name_t *name)*;

DESCRIPTION

A **do_mach_notify_dead_name** function is called as the result of a kernel message indicating that a message forcibly queued to a port via MACH_NOTIFY_SEND was accepted. *notify* is the port named via **mach_msg**.

(Note: This feature is current planned for deletion.)

PARAMETERS

notify
The port to which the notification was sent.

name
The port whose message was accepted.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_dead_name**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_no_senders

do_mach_notify_no_senders Server Interface — Handles the occurrence of a no-more-senders notification

SYNOPSIS

kern_return_t **do_mach_notify_no_senders** (notify_port_t *notify,*
mach_port_mscount_t *mscount)*;

DESCRIPTION

A **do_mach_notify_no_senders** function is called as the result of a kernel message indicating that a receive right has no more senders. *notify* is the port named via **mach_port_request_notification**.

PARAMETERS

notify
The port to which the notification was sent.

mscount
The value the port's make-send count had when it was generated.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_dead_name**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once.**

do_mach_notify_port_destroyed

do_mach_notify_port_destroyed Server Interface — Handles the occurrence of a port destroyed notification

SYNOPSIS

kern_return_t **do_mach_notify_port_destroyed** (notify_port_t *notify,*
mach_port_receive_t *rights)*;

DESCRIPTION

A **do_mach_notify_port_destroyed** function is called as the result of a kernel message indicating that a receive right would have been destroyed. *notify* is the port named via **mach_port_request_notification**.

(Note: This feature is currently planned for deletion.)

PARAMETERS

notify
The port to which the notification was sent.

rights
The receive right that would have been destroyed.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_dead_name**, **do_mach_notify_send_once.**

do_mach_notify_send_once

do_mach_notify_send_once Server Interface — Handles the occurrence of a send-once notification

SYNOPSIS

kern_return_t **do_mach_notify_send_once** (notify_port_t *notify)*;

DESCRIPTION

A **do_mach_notify_send_once** function is called as the result of a kernel message indicating that a send-once right was destroyed. *notify* is the port named via **mach_msg**.

PARAMETERS

notify
The port to which the notification was sent.

RETURN VALUE

KERN_SUCCESS
The notification was received.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_dead_name.**

mach_port_allocate

mach_port_allocate Function — Creates a port right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_allocate** (ipc_space_t *task,* mach_port_right_t *right,* mach_port_t **name);*

DESCRIPTION

The **mach_port_allocate** function creates a new right in the specified task. The new right's name is returned in *name*.

PARAMETERS

task

The task acquiring the port right.

right

The kind of right which will be created. This is one of the following:

MACH_PORT_RIGHT_RECEIVE

mach_port_allocate creates a port. The new port is not a member of any port set. It doesn't have any extant send or send-once rights. Its make-send count is zero, its queue limit is MACH_PORT_QLIMIT_DEFAULT, and it has no queued messages. *name* denotes the receive right for the new port.

task does not hold send rights for the new port, only the receive right. **mach_port_insert_right** and **mach_port_extract_right** can be used to convert the receive right into a combined send/receive right.

MACH_PORT_RIGHT_PORT_SET

mach_port_allocate creates a port set. The new port set has no members.

MACH_PORT_RIGHT_DEAD_NAME

mach_port_allocate creates a dead name. The new dead name has one user reference.

name
: [out] The task's name for the port right. This can be any name that wasn't in use.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
: *right* was invalid.

KERN_NO_SPACE
: There was no room in *task*'s IPC name space for another right.

KERN_RESOURCE_SHORTAGE
: The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_allocate_name**, **mach_port_deallocate, mach_-port_insert_right, mach_port_extract_right**.

mach_port_allocate_name

mach_port_allocate_name Function — Creates a port right with a given name

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_allocate_name**(ipc_space_t *task,*
mach_port_right_t *right*, mach_port_t *name*);

DESCRIPTION

The **mach_port_allocate_name** function creates a new right in the specified task, with a specified name for the new right.

PARAMETERS

task
The task acquiring the port right.

right
The kind of right which will be created. This is one of the following values:

MACH_PORT_RIGHT_RECEIVE
mach_port_allocate_name creates a port. The new port is not a member of any port set. It doesn't have any extant send or send-once rights. Its make-send count is zero, its queue limit is MACH_PORT_QLIMIT_DEFAULT, and it has no queued messages. *name* denotes the receive right for the new port.

task does not hold send rights for the new port, only the receive right. **mach_port_insert_right** and **mach_port_extract_right** can be used to convert the receive right into a combined send/receive right.

MACH_PORT_RIGHT_PORT_SET
mach_port_allocate_name creates a port set. The new port set has no members.

MACH_PORT_RIGHT_DEAD_NAME
mach_port_allocate_name creates a new dead name. The new dead name has one user reference.

name
The task's name for the port right. *name* must not already be in use for some right, and it can't be the reserved values MACH_PORT_-NULL and MACH_PORT_DEAD.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
right was invalid.

KERN_INVALID_VALUE
name was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
name was already in use for a port right.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_deallocate**, **mach_port_rename**.

mach_port_deallocate

mach_port_deallocate Function — Releases a user reference for a right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_deallocate**(ipc_space_t *task,* mach_port_t *name);*

DESCRIPTION

The **mach_port_deallocate** function releases a user reference for a right. It is an alternate form of **mach_port_mod_refs** that allows a task to release a user reference for a send or send-once right without failing if the port has died and the right is now actually a dead name.

If *name* denotes a dead name, send right, or send-once right, then the right loses one user reference. If it only had one user reference, then the right is destroyed.

PARAMETERS

task
: The task holding the right.

name
: The task's name for the right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted an invalid right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name, mach_port_-mod_refs**.

mach_port_destroy

mach_port_destroy Function — Removes a task's rights for a name

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_destroy** (ipc_space_t *task*; mach_port_t *name)*;

DESCRIPTION

The **mach_port_destroy** function de-allocates all rights denoted by a name. The name becomes immediately available for reuse.

For most purposes, **mach_port_mod_refs** and **mach_port_deallocate** are preferable.

If *name* denotes a port set, then all members of the port set are implicitly removed from the port set.

If *name* denotes a receive right that is a member of a port set, the receive right is implicitly removed from the port set. If there is a port-destroyed request registered for the port, then the receive right is not actually destroyed, but instead is sent in a port-destroyed notification. (Note: Port destroyed notifications are currently planned for deletion.) If there is no registered port-destroyed request, remaining messages queued to the port are destroyed and extant send and send-once rights turn into dead names. If those send and send-once rights have dead-name requests registered, then dead-name notifications are generated for them.

If *name* denotes a send-once right, then the send-once right is used to produce a send-once notification for the port.

If *name* denotes a send-once, send, and/or receive right, and it has a dead-name request registered, then the registered send-once right is used to produce a port-deleted notification for the name.

PARAMETERS

task
: The task holding the right.

name
: The task's name for the right.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_allocate**, **mach_port_allocate_name,** **mach_port_-mod_refs**, **mach_port_deallocate**, **mach_port_request_notification**.

mach_port_extract_right

mach_port_extract_right Function — Extracts a port right from a task

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_extract_right**(ipc_space_t *task,* mach_port_t *name,* mach_msg_type_name_t *desired_type,* mach_port_t **right,* mach_msg_type_name_t **acquired_type)*;

DESCRIPTION

The **mach_port_extract_right** function extracts a port right from the target task and returns it to the caller as if the task sent the right voluntarily, using *desired_type* as the value of *msgt_name*. See **mach_msg**.

The returned value of *acquired_type* will be MACH_MSG_TYPE_PORT_S-END if a send right is extracted, MACH_MSG_TYPE_PORT_RECEIVE if a receive right is extracted, and MACH_MSG_TYPE_PORT_SEND_ONCE if a send-once right is extracted.

PARAMETERS

task
: The task holding the port right.

name
: The task's name for the port right.

desired_type
: IPC type, specifying how the right should be extracted.

right
: [out] The extracted right.

acquired_type
: [out] The type of the extracted right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

KERN_INVALID_VALUE
desired_type was invalid.

RELATED INFORMATION

Functions: **mach_port_insert_right**, **mach_msg**.

mach_port_get_receive_status

mach_port_get_receive_status Function — Returns the status of a receive right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_get_receive_status**(ipc_space_t *task,* mach_port_t *name,* mach_port_status_t **status)*;

DESCRIPTION

The **mach_port_get_receive_status** function returns the current status of the specified receive right.

PARAMETERS

task
The task holding the receive right.

name
The task's name for the receive right.

status
[out] The status information for the receive right.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted a right, but not a receive right.

RELATED INFORMATION

Functions: **mach_port_set_qlimit**, **mach_port_set_mscount**.

mach_port_get_refs

mach_port_get_refs Function — Retrieves the number of user references for a right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_get_refs** (ipc_space_t *task,* mach_port_t *name,*
mach_port_right_t *right,* mach_port_urefs_t **refs)*;

DESCRIPTION

The **mach_port_get_refs** function returns the number of user references a task has for a right.

If *name* denotes a right, but not the type of right specified, then zero is returned. Otherwise a positive number of user references is returned. Note a name may simultaneously denote send and receive rights.

PARAMETERS

task
: The task holding the right.

name
: The task's name for the right.

right
: The type of right being examined: MACH_PORT_RIGHT_SEND, MACH_PORT_RIGHT_RECEIVE, MACH_PORT_RIGHT_SEND_ONCE, MACH_PORT_RIGHT_PORT_SET or MACH_PORT_RIGHT_DEAD_NAME.

refs
: [out] Number of user references.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
: *right* was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_mod_refs**.

mach_port_get_set_status

mach_port_get_set_status Function — Returns the members of a port set

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_get_set_status**(ipc_space_t *task,* mach_port_t *name,* mach_port_array_t **members,* mach_msg_type_number_t **count)*;

DESCRIPTION

The **mach_port_get_set_status** function returns the members of a port set. *members* is an array that is automatically allocated when the reply message is received.

PARAMETERS

task
: The task holding the port set.

name
: The task's name for the port set.

members
: [out array] The task's names for the port set's members.

count
: [out] The number of member names returned.

NOTES

members is allocated by the kernel, as if by **vm_allocate**. The user should **vm_deallocate** it when the data is no longer needed

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted a right, but not a port set.

KERN_RESOURCE_SHORTAGE
: The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_move_member**, **vm_deallocate**.

mach_port_insert_right

mach_port_insert_right Function — Inserts a port right into a task

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_insert_right**(ipc_space_t *task,* mach_port_t *name,* mach_port_t *right,* mach_msg_type_name_t *right_type)*;

DESCRIPTION

The **mach_port_insert_right** function inserts into *task* the caller's right for a port, using a specified name for the right in the target task.

The specified *name* can't be one of the reserved values MACH_PORT_-NULL or MACH_PORT_DEAD. The right can't be MACH_PORT_NULL or MACH_PORT_DEAD.

The argument *right_type* specifies a right to be inserted and how that right should be extracted from the caller. It should be a value appropriate for *msgt_-name*; see **mach_msg**.

If *right_type* is MACH_MSG_TYPE_MAKE_SEND, MACH_MSG_TYPE_-MOVE_SEND, or MACH_MSG_TYPE_COPY_SEND, then a send right is inserted. If the target already holds send or receive rights for the port, then *name* should denote those rights in the target. Otherwise, *name* should be un-used in the target. If the target already has send rights, then those send rights gain an additional user reference. Otherwise, the target gains a send right, with a user reference count of one.

If *right_type* is MACH_MSG_TYPE_MAKE_SEND_ONCE or MACH_MS-G_TYPE_MOVE_SEND_ONCE, then a send-once right is inserted. The *name* should be unused in the target. The target gains a send-once right.

If *right_type* is MACH_MSG_TYPE_MOVE_RECEIVE, then a receive right is inserted. If the target already holds send rights for the port, then *name* should denote those rights in the target. Otherwise, *name* should be unused in the target. The receive right is moved into the target task.

PARAMETERS

task

The task which gets the caller's right.

name
The name by which *task* will know the right.

right
The port right.

right_type
IPC type of the sent right; e.g., MACH_MSG_TYPE_COPY_SEND or MACH_MSG_TYPE_MOVE_RECEIVE.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
name was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
name already denoted a right.

KERN_INVALID_VALUE
right was not a port right.

KERN_INVALID_CAPABILITY
right was null or dead.

KERN_UREFS_OVERFLOW
Inserting the right would overflow *name*'s user-reference count.

KERN_RIGHT_EXISTS
task already had rights for the port, with a different name.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_extract_right**, **mach_msg**.

mach_port_mod_refs

mach_port_mod_refs Function — Changes the number of user refs for a right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_mod_refs** (ipc_space_t *task,* mach_port_t *name,*
mach_port_right_t *right,* mach_port_delta_t *delta)*;

DESCRIPTION

The **mach_port_mod_refs** function requests that the number of user references a task has for a right be changed. This results in the right being destroyed, if the number of user references is changed to zero.

The *name* should denote the specified right. The number of user references for the right is changed by the amount *delta*, subject to the following restrictions: port sets, receive rights, and send-once rights may only have one user reference. The resulting number of user references can't be negative. If the resulting number of user references is zero, the effect is to de-allocate the right. For dead names and send rights, there is an implementation-defined maximum number of user references.

If the call destroys the right, then the effect is as described for **mach_port_destroy**, with the exception that **mach_port_destroy** simultaneously destroys all the rights denoted by a name, while **mach_port_mod_refs** can only destroy one right. The name will be available for reuse if it only denoted the one right.

PARAMETERS

task
: The task holding the right.

name
: The task's name for the right.

right
: The type of right being modified: MACH_PORT_RIGHT_SEND, MACH_PORT_RIGHT_RECEIVE, MACH_PORT_RIGHT_SEND_ONCE, MACH_PORT_RIGHT_PORT_SET or MACH_PORT_RIGHT_DEAD_NAME.

delta
: Signed change to the number of user references.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_VALUE
: *right* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted a right, but not the specified right.

KERN_INVALID_VALUE
: The user-reference count would become negative.

KERN_UREFS_OVERFLOW
: The user-reference count would overflow.

RELATED INFORMATION

Functions: **mach_port_destroy**, **mach_port_get_refs**.

mach_port_move_member

mach_port_move_member Function — Moves a receive right into/out of a port set

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_move_member**(ipc_space_t *task,* mach_port_t *member,* mach_port_t *after)*;

DESCRIPTION

The **mach_port_move_member** function moves a receive right into a port set. If the receive right is already a member of another port set, it is removed from that set first. If the port set is MACH_PORT_NULL, then the receive right is not put into a port set, but removed from its current port set.

PARAMETERS

task
: The task holding the port set and receive right.

member
: The task's name for the receive right.

after
: The task's name for the port set.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *member* did not denote a right.

KERN_INVALID_RIGHT
: *member* denoted a right, but not a receive right.

KERN_INVALID_NAME
: *after* did not denote a right.

KERN_INVALID_RIGHT
: *after* denoted a right, but not a port set.

KERN_NOT_IN_SET
: *after* was MACH_PORT_NULL, but *member* wasn't currently in a port set.

RELATED INFORMATION

Functions: **mach_port_get_set_status**, **mach_port_get_receive_status**.

mach_port_names

mach_port_names Function — Return information about a task's port name space

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_names** (ipc_space_t *task,* mach_port_array_t **names,* mach_msg_type_number_t **ncount,* mach_port_type_array_t **types,* mach_msg_type_number_t **tcount)*;

DESCRIPTION

The **mach_port_names** returns information about *task*'s port name space. It returns *task*'s currently active names, which represent some port, port set, or dead name right. For each name, it also returns what type of rights *task* holds. (The same information returned by **mach_port_type**.)

PARAMETERS

task
: The task whose port name space is queried.

names
: [out array] The names of the ports, port sets, and dead names in the task's port name space, in no particular order.

ncount
: [out] The number of names returned.

types
: [out array] The type of each corresponding name. Indicates what kind of rights the task holds with that name.

tcount
: [out] Should be the same as *ncount*.

NOTES

names and *types* are arrays that are automatically allocated when the reply message is received, as if by **vm_allocate**. The user should **vm_deallocate** them when the data is no longer needed.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_type**, **vm_deallocate**.

mach_port_rename

mach_port_rename Function — Change a task's name for a right

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_rename** (ipc_space_t *task,* mach_port_t *old_name,*
mach_port_t *new_name)*;

DESCRIPTION

The **mach_port_rename** function changes the name by which a port, port set, or dead name is known to *task*. *new_name* must not already be in use, and it can't be the distinguished values MACH_PORT_NULL and MACH_-PORT_DEAD.

PARAMETERS

task
: The task holding the port right.

old_name
: The original name of the port right.

new_name
: The new name for the port right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *old_name* did not denote a right.

KERN_INVALID_VALUE
: *new_name* was MACH_PORT_NULL or MACH_PORT_DEAD.

KERN_NAME_EXISTS
: *new_name* already denoted a right.

KERN_RESOURCE_SHORTAGE
: The kernel ran out of memory.

RELATED INFORMATION

Functions: **mach_port_names**.

mach_port_request_notification

mach_port_request_notification Function — Request a notification of a port event

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_request_notification**(ipc_space_t *task,*
mach_port_t *name,* mach_msg_id_t *variant,*
mach_port_mscount_t *sync,* mach_port_t *notify,*
mach_msg_type_name_t *notify_type,* mach_port_t
**previous)*;

DESCRIPTION

The **mach_port_request_notification** function registers a request for a notification and supplies a send-once right that the notification will use. It is an atomic swap, returning the previously registered send-once right (or MACH_-PORT_NULL for none). A notification request may be cancelled by providing MACH_PORT_NULL.

The *variant* argument takes the following values:

MACH_NOTIFY_PORT_DESTROYED
: *sync* must be zero. The *name* must specify a receive right, and the call requests a port-destroyed notification for the receive right. If the receive right were to have been destroyed, say by **mach_port_destroy**, then instead the receive right will be sent in a port-destroyed notification to the registered send-once right.

(Note: This feature is currently planned for deletion.)

MACH_NOTIFY_DEAD_NAME
: The call requests a dead-name notification. *name* specifies send, receive, or send-once rights for a port. If the port is destroyed (and the right remains, becoming a dead name), then a dead-name notification which carries the name of the right will be sent to the registered send-once right. If *sync* is non-zero, the *name* may specify a dead name, and a dead-name notification is immediately generated.

Whenever a dead-name notification is generated, the user reference count of the dead name is incremented. For example, a send right with two user refs has a registered dead-name request. If the port is destroyed, the send right turns into a dead name with three user refs (instead of two), and a dead-name notification is generated.

If the name is made available for reuse, perhaps because of **mach_-port_destroy** or **mach_port_mod_refs**, or the name denotes a send-once right which has a message sent to it, then the registered send-once right is used to generate a port-deleted notification.

MACH_NOTIFY_NO_SENDERS

The call requests a no-senders notification. *name* must specify a receive right. If the receive right's make-send count is greater than or equal to the sync value, and it has no extant send rights, than an immediate no-senders notification is generated. Otherwise the notification is generated when the receive right next loses its last extant send right. In either case, any previously registered send-once right is returned.

The no-senders notification carries the value the port's make-send count had when it was generated. The make-send count is incremented whenever MACH_MSG_TYPE_MAKE_SEND is used to create a new send right from the receive right. The make-send count is reset to zero when the receive right is carried in a message.

(Note: Currently, moving a receive right does not affect any extant no-senders notifications. It is currently planned to change this so that no-senders notifications are canceled, with a send-once notification sent to indicate the cancelation.)

PARAMETERS

task

The task holding the specified right.

name

The task's name for the right.

variant

The type of notification.

sync

Some variants use this value to overcome race conditions.

notify

A send-once right, to which the notification will be sent.

notify_type

IPC type of the sent right; either MACH_MSG_TYPE_MAKE_-SEND_ONCE or MACH_MSG_TYPE_MOVE_SEND_ONCE.

previous

[out] The previously registered send-once right.

RETURN VALUE

KERN_SUCCESS

The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_VALUE
variant was invalid.

KERN_INVALID_NAME
name did not denote a right.

KERN_INVALID_RIGHT
name denoted an invalid right.

KERN_INVALID_CAPABILITY
notify was invalid.

When using MACH_NOTIFY_PORT_DESTROYED:

KERN_INVALID_VALUE
sync was not zero.

When using MACH_NOTIFY_DEAD_NAME:

KERN_RESOURCE_SHORTAGE
The kernel ran out of memory.

KERN_INVALID_ARGUMENT
name denotes a dead name, but *sync* is zero or *notify* is null.

KERN_UREFS_OVERFLOW
name denotes a dead name, but generating an immediate dead-name notification would overflow the name's user-reference count.

RELATED INFORMATION

Functions: **mach_port_get_receive_status**.

Data structures: **mach_dead_name_notification**, **mach_port_destroyed_notification**, **mach_no_senders_notification**.

mach_port_set_mscount

mach_port_set_mscount Function — Changes the make-send count of a port

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_set_mscount**(ipc_space_t *task,* mach_port_t *name,* mach_port_mscount_t *mscount)*;

DESCRIPTION

The **mach_port_set_mscount** function changes the make-send count of *task*'s receive right named *name*. All values for *mscount* are valid.

PARAMETERS

task
: The task owning the receive right.

name
: *task*'s name for the receive right.

mscount
: New value for the make-send count for the receive right.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted a right, but not a receive right.

RELATED INFORMATION

Functions: **mach_port_get_receive_status**, **mach_port_set_qlimit**.

mach_port_set_qlimit

mach_port_set_qlimit Function — Changes the queue limit of a port

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_set_qlimit**(ipc_space_t *task,* mach_port_t *name,*
mach_port_msgcount_t *qlimit)*;

DESCRIPTION

The **mach_port_set_qlimit** function changes the queue limit of *task*'s receive right named *name*. Valid values for *qlimit* are between zero and MACH_-PORT_QLIMIT_MAX, inclusive.

PARAMETERS

task
: The task owning the receive right.

name
: *task*'s name for the receive right.

qlimit
: The number of messages which may be queued to this port without causing the sender to block.

RETURN VALUE

KERN_SUCCESS
: The call succeeded.

KERN_INVALID_TASK
: *task* was invalid.

KERN_INVALID_NAME
: *name* did not denote a right.

KERN_INVALID_RIGHT
: *name* denoted a right, but not a receive right.

KERN_INVALID_VALUE
: *qlimit* was invalid.

RELATED INFORMATION

Functions: **mach_port_get_receive_status**, **mach_port_set_mscount**.

mach_port_type

mach_port_type Function — Return information about a task's port name

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_port.h>**)

SYNOPSIS

kern_return_t **mach_port_type** (ipc_space_t *task,* mach_port_t *name,*
mach_port_type_t **ptype)*;

DESCRIPTION

The **mach_port_type** function returns information about *task*'s rights for a specific name in its port name space. The returned *ptype* is a bit-mask indicating what rights *task* holds with this name. The bit-mask is composed of the following bits:

MACH_PORT_TYPE_SEND
The name denotes a send right.

MACH_PORT_TYPE_RECEIVE
The name denotes a receive right.

MACH_PORT_TYPE_SEND_ONCE
The name denotes a send-once right.

MACH_PORT_TYPE_PORT_SET
The name denotes a port set.

MACH_PORT_TYPE_DEAD_NAME
The name is a dead name.

MACH_PORT_TYPE_DNREQUEST
A dead-name request has been registered for the right.

MACH_PORT_TYPE_MAREQUEST
A msg-accepted request for the right is pending. (Note: This feature is planned for deletion.)

MACH_PORT_TYPE_COMPAT
The port right was created in the compatibility mode.

PARAMETERS

task
The task whose port name space is queried.

name
The name being queried.

ptype
[out] The type of the name. Indicates what kind of right the task holds for the port, port set, or dead name.

RETURN VALUE

KERN_SUCCESS
The call succeeded.

KERN_INVALID_TASK
task was invalid.

KERN_INVALID_NAME
name did not denote a right.

RELATED INFORMATION

Functions: **mach_port_names**, **mach_port_get_receive_status**, **mach_-port_get_set_status**.

mach_ports_lookup

mach_ports_lookup Function — Returns an array of well-known system ports.

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **mach_ports_lookup** (task_t *target_task,* mach_port_array_t **init_port_set,* mach_msg_type_number_t **init_port_count)*;

DESCRIPTION

The **mach_ports_lookup** function returns an array of the well-known system ports that are currently registered for the specified task. Note that the task holds only send rights for the ports.

Registered ports are those ports that are used by the run-time system to initialize a task. To register system ports for a task, use the **mach_ports_register** function.

PARAMETERS

target_task
: The task whose currently registered ports are to be returned.

init_port_set
: [out] The returned array of ports.

init_port_count
: [out] The number of ports in the array.

RETURN VALUE

KERN_SUCCESS
: The array of registered ports has been returned.

RELATED INFORMATION

Functions: **mach_ports_register**.

mach_ports_register

mach_ports_register Function — Registers an array of well-known system ports

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **mach_ports_register**(task_t *target_task,* mach_port_array_t *init_port_set,* mach_msg_type_number_t *init_port_array_count)*;

DESCRIPTION

The **mach_ports_register** function registers an array of well-known system ports for the specified task. The task holds only send rights for the registered ports. The valid well-known system ports are:

- The port for the Network Name Server.
- The port for the Environment Manager.
- The port for the Service server.

Each port must be placed in a specific slot in the array. The slot numbers are defined (in **mach.h**) by the global constants NAME_SERVER_SLOT, ENVIRONMENT_SLOT, and SERVICE_SLOT.

A task can retrieve the currently registered ports by using the **mach_ports_lookup** function.

PARAMETERS

target_task
: The task for which the ports are to be registered.

init_port_set
: The array of ports to register.

init_port_array_count
: The number of ports in the array. Note that while this is a variable, the kernel accepts only a limited number of ports. The maximum number of ports is defined by the global constant MACH_PORT_SLOTS_USED.

NOTES

When a new task is created (with **task_create**), the child task can inherit the parent's registered ports. Note that child tasks do not automatically acquire rights to these ports. They must use **mach_ports_lookup** to get them. It is intended that port registration be used only for task initialization, and then only by run-time support modules.

A parent task has three choices when passing registered ports to child tasks:

- The parent task can do nothing. In this case, all child tasks inherit access to the same ports that the parent has.
- The parent task can use **mach_ports_register** to modify its set of registered ports before creating child tasks. In this case, the child tasks get access to the modified set of ports. After creating its child tasks. the parent can use **mach_ports_register** again to reset its registered ports.
- The parent task can first create a specific child task and then use **mach_ports_register** to modify the child's inherited set of ports, before starting the child's thread(s). The parent must specify the child's task port, rather than its own, on the call to **mach_ports_register**.

Tasks other than the Network Name Server and the Environment Manager should not need access to the Service port. The Network Name Server port is the same for all tasks on a given machine. The Environment port is the only port likely to have different values for different tasks.

Registered ports are restricted to those ports that are used by the run-time system to initialize a task. A parent task can pass other ports to its child tasks through:

- An initial message (see **mach_msg**).
- The Network Name Server, for public ports.
- The Environment Manager, for private ports.

RETURN VALUE

KERN_SUCCESS
: The ports have been registered for the task.

KERN_INVALID_ARGUMENT
: The number of ports exceeds the allowed maximum.

RELATED INFORMATION

Functions: **mach_msg**, **mach_ports_lookup**.

mach_reply_port

mach_reply_port Function — Creates a port for the task

LIBRARY

libmach.a

#include <mach/mach_traps.h>

SYNOPSIS

mach_port_t **mach_reply_port**();

DESCRIPTION

The **mach_reply_port** function creates a new port for the current task and returns the name assigned by the kernel. The kernel records the name in the task's port name space and grants the task receive rights for the port. The new port is not a member of any port set.

This function is an optimized version of **mach_port_allocate** that uses no port references. Its main purpose is to allocate a reply port for the task when the task is starting— namely, before it has any ports to use as reply ports for any IPC based system functions.

PARAMETERS

None

NOTES

This function is actually a system call.

CAUTIONS

Although the created port can be used for any purpose, the implementation may optimize its use as a reply port.

RETURN VALUE

MACH_PORT_NULL
: No port was allocated. Any other value indicates success.

RELATED INFORMATION

Functions: **mach_port_allocate**.

notify_server

notify_server Function — Handle kernel generated IPC notifications

LIBRARY

libmach.a

SYNOPSIS

boolean_t **notify_server** (mach_msg_header_t **in_msg,* mach_msg_header_t **out_msg*);

DESCRIPTION

The **notify_server** function is the MIG generated server handling function to handle messages from the kernel corresponding to IPC notifications. Such messages are delivered to the notification port named in a **mach_msg** or **mach_port_request_notification** call. The **notify_server** function performs all necessary argument handling for this kernel message and calls the appropriate handling function. These functions must be supplied by the caller; their names are: **do_mach_notify_dead_name**, **do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once**.

PARAMETERS

in_msg
: The notification message received from the kernel.

out_msg
: Not used.

RETURN VALUE

TRUE
: The message was handled and the appropriate function was called.

FALSE
: The message did not apply to the notification mechanism and no other action was taken.

RELATED INFORMATION

Functions: **mach_msg**, **mach_port_request_notification, do_mach_notify_dead_name**, **do_mach_notify_msg_accepted**, **do_mach_notify_no_senders**, **do_mach_notify_port_destroyed**, **do_mach_notify_send_once**.

CHAPTER 4

Virtual Memory Interface Specifications

This chapter discusses the specifics of the kernel's virtual memory interfaces. This includes memory status related functions associated with a single task. Functions that are related to, or used by, external memory managers (pagers) are described in the next chapter.

vm_allocate

vm_allocate Function — Allocates a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_allocate** (vm_task_t *target_task,* vm_address_t **address,*
vm_size_t *size,* boolean_t *anywhere)*;

DESCRIPTION

The **vm_allocate** function allocates a region of virtual memory in the specified task's address space. A new region is always zero filled. The physical memory is not allocated until an executing thread references the new virtual memory.

If *anywhere* is true, the returned *address* will be at a page boundary and *size* will be rounded up to an integral number of pages. Otherwise, the region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory allocated may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

Use the **mach_task_self** function to return the caller's value for *target_task*. This macro returns the task kernel port for the caller.

Initially, there are no access restrictions on any of the pages of the newly allocated region. Child tasks inherit the new region as a copy.

To establish different protections for the new region, use the **vm_protect** and **vm_inherit** functions.

PARAMETERS

target_task
: The task in whose address space the region is to be allocated.

address
: [in/out] The starting address for the region. If there is not enough room following the address, the kernel does not allocate the region. The kernel returns the starting address actually used for the allocated region.

size
: The number of bytes to allocate.

anywhere
: Placement indicator. If false, the kernel allocates the region starting at *address*. If true, the kernel allocates the region wherever enough space is available within the address space. The kernel returns the starting address actually used in *address*.

NOTES

For languages other than C, use the **vm_statistics** and **mach_task_self** functions to return the task's kernel port (for *target_task*).

A region is a continuous range of addresses bounded by a start address and an end address. Regions consist of pages that have different protection or inheritance characteristics.

A task's address space can contain both explicitly allocated memory and automatically allocated memory. The **vm_allocate** function explicitly allocates memory. The kernel automatically allocates memory to hold out-of-line data passed in a message (and received with **mach_msg**). The kernel allocates memory for the passed data as an integral number of pages.

RETURN VALUE

KERN_SUCCESS
: The new region has been allocated.

KERN_INVALID_ADDRESS
: The specified address is illegal.

KERN_NO_SPACE
: There is not enough space in the task's address space to allocate the new region.

RELATED INFORMATION

Functions: **task_get_special_port**, **vm_deallocate**, **vm_inherit**, **vm_protect**, **vm_region**, **vm_statistics**.

vm_copy

vm_copy Function — Copies a region in a task's virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_copy** (vm_task_t *target_task,* vm_address_t *source_address*, vm_size_t *count,* vm_address_t *dest_address)*;

DESCRIPTION

The **vm_copy** function copies a source region to a destination region within a task's virtual memory. The destination region can overlap the source region.

The destination region must already be allocated. The source region must be readable, and the destination region must be writable.

PARAMETERS

target_task
: The task whose memory is to be copied.

source_address
: The starting address for the source region. The address must be on a page boundary.

count
: The number of bytes in the source region. The number of bytes must convert to an integral number of virtual pages.

dest_address
: The starting address for the destination region. The address must be on a page boundary.

RETURN VALUE

KERN_SUCCESS
: The memory region has been copied.

KERN_INVALID_ARGUMENT
: Either an address does not start on a page boundary or the count does not convert to an integral number of pages.

KERN_PROTECTION_FAILURE
: The source region is protected against reading, or the destination region is protected against writing.

KERN_INVALID_ADDRESS
An address is illegal or specifies a non-allocated region, or there is not enough memory following one of the addresses.

RELATED INFORMATION

Functions: **vm_protect**, **vm_write**, **vm_statistics**.

vm_deallocate

vm_deallocate Function — De-allocates a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_deallocate** (vm_task_t *target_task,* vm_address_t *address,*
vm_size_t *size)*;

DESCRIPTION

The **vm_deallocate** function de-allocates a region of virtual memory in the specified task's address space.

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory de-allocated may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

vm_deallocate can be used to de-allocate memory passed as out-of-line data in a message.

vm_deallocate affects only *target_task*. Other tasks that have access to the de-allocated memory can continue to reference it.

PARAMETERS

target_task
: The task in whose address space the region is to be de-allocated.

address
: The starting address for the region.

size
: The number of bytes to de-allocate.

RETURN VALUE

KERN_SUCCESS
: The region has been de-allocated.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

Functions: **mach_msg**, **vm_allocate**, **vm_statistics**.

vm_inherit

vm_inherit Function — Sets the inheritance attribute for a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_inherit** (vm_task_t *target_task,* vm_address_t *address,*
vm_size_t *size,* vm_inherit_t *new_inheritance)*;

DESCRIPTION

The **vm_inherit** function sets the inheritance attribute for a region within the specified task's address space. The inheritance attribute determines the type of access established for child tasks at task creation

Because inheritance applies to virtual pages, the specified *address* and *size* are rounded to page boundaries, as follows: the region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory affected may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

A parent and a child task can share the same physical memory only if the inheritance for the memory is set to VM_INHERIT_SHARE before the child task is created. This is the only way that two tasks can share memory (other than through the use of an external memory manager; see **vm_map**).

Note that all the threads within a task share the task's memory.

PARAMETERS

target_task
The task whose address space contains the region.

address
The starting address for the region.

size
The number of bytes in the region.

new_inheritance
The new inheritance attribute for the region. Valid values are:

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
Gives child tasks a copy of the region.

VM_INHERIT_NONE
Provides no access to the region for child tasks.

RETURN VALUE

KERN_SUCCESS
The new inheritance has been set for the region.

KERN_INVALID_ADDRESS
The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

Functions: **task_create**, **vm_map**, **vm_region**.

vm_machine_attribute

vm_machine_attribute Function — Sets and gets special attributes of a memory region

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_machine_attribute**(vm_task_t *target_task,* vm_address_t *address,* vm_size_t *size,* vm_machine_attribute_t *attribute,* vm_machine_attribute_val_t **value)*;

DESCRIPTION

The **vm_machine_attribute** function gets and sets special attributes of the memory region implemented by the implementations underlying **pmap** module. These attributes are properties such as cachability, migrability and replicability. The behavior of this function is machine dependent.

PARAMETERS

target_task
: The task in whose address space the memory object is to be manipulated.

address
: The starting address for the memory region. The granularity of rounding of this value to page boundaries is implementation dependent.

size
: The number of bytes in the region. The granularity of rounding of this value to page boundaries is implementation dependent.

attribute
: The name of the attribute to be get/set. Possible values are:

MATTR_CACHE
: Cachability

MATTR_MIGRATE
: Migratability

MATTR_REPLICATE
: Replicability

value

[in/out] The new value for the attribute. The old value is also returned in this variable.

MATTR_VAL_OFF
(generic) turn attribute off

MATTR_VAL_ON
(generic) turn attribute on

MATTR_VAL_GET
(generic) return current value

MATTR_VAL_CACHE_FLUSH
flush from all caches

MATTR_VAL_DCACHE_FLUSH
flush from data caches

MATTR_VAL_ICACHE_FLUSH
flush from instruction caches

RETURN VALUE

KERN_SUCCESS
The memory object has been modified.

KERN_INVALID_ARGUMENT
An illegal argument was specified.

vm_map

vm_map Function — Maps a memory object to a task's address space

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_map** (vm_task_t *target_task,* vm_address_t **address,* vm_size_t *size,* vm_address_t *mask,* boolean_t *anywhere,* memory_object_t *memory_object,* vm_offset_t *offset,* boolean_t *copy,* vm_prot_t *cur_protection,* vm_prot_t *max_protection,* vm_inherit_t *inheritance)*;

DESCRIPTION

The **vm_map** function maps a portion of the specified memory object into the virtual address space belonging to *target_task*. The target task can be the calling task or another task, identified by its task kernel port.

The portion of the memory object mapped is determined by *offset* and *size*. The kernel maps *address* to the offset, so that an access to the memory starts at the offset in the object.

The *mask* parameter specifies additional alignment restrictions on the kernel's selection of the starting address. Uses for this mask include:

- Forcing the memory address alignment for a mapping to be the same as the alignment within the memory object.
- Quickly finding the beginning of an allocated region by performing bit arithmetic on an address known to be in the region.
- Emulating a larger virtual page size.

The *cur_protection*, *max_protection*, and *inheritance* parameters set the protection and inheritance attributes for the mapped object. As a rule, at least the maximum protection should be specified so that a server can make a restricted (for example, read-only) mapping in a client atomically. The current protection and inheritance parameters are provided for convenience so that the caller does not have to call **vm_inherit** and **vm_protect** separately.

The same memory object can be mapped in more than once and by more than one task. If an object is mapped by multiple tasks, the kernel maintains consistency for all the mappings if they use the same page alignment for *offset* and are on the same host. In this case, the virtual memory to which the object is mapped is shared by all the tasks. Changes made by one task in its address space are visible to all the other tasks.

PARAMETERS

target_task
: The task to whose address space the memory object is to be mapped.

address
: [in/out] The starting address for the mapped object. If the address is not at the beginning of a virtual page, the kernel rounds it up to the next page boundary. If there is not enough room following the address, the kernel does not map the object. The kernel returns the starting address actually used for the mapped object.

size
: The number of bytes to allocate for the object. The kernel rounds this number up to an integral number of virtual pages.

mask
: Alignment restrictions for starting address. Bits turned on in the mask cannot be turned on in the starting address.

anywhere
: Placement indicator. If false, the kernel allocates the object's region starting at *address*. If true, the kernel allocates the region anywhere at or following *address* that there is enough space available within the address space. The kernel returns the starting address actually used in *address*.

memory_object
: The port naming the abstract memory object. If MEMORY_OBJECT_NULL is specified, the kernel allocates zero-filled memory, as with **vm_allocate**.

offset
: An offset within the memory object, in bytes. The kernel maps *address* to the specified offset.

copy
: Copy indicator. If true, the kernel copies the region for the memory object to the specified task's address space. If false, the region is mapped read-write.

cur_protection
: The initial current protection for the region. Valid values are obtained by or'ing together the following values:

  VM_PROT_READ
  : Allows read access.

  VM_PROT_WRITE
  : Allows write access.

VM_PROT_EXECUTE
Allows execute access.

max_protection
The maximum protection for the region. Values are the same as for *cur_protection*.

inheritance
The initial inheritance attribute for the region. Valid values are:

VM_INHERIT_SHARE
Allows child tasks to share the region.

VM_INHERIT_COPY
Gives child tasks a copy of the region.

VM_INHERIT_NONE
Provides no access to the region for child tasks.

NOTES

vm_map allocates a region in a task's address space and maps the specified memory object to this region. **vm_allocate** allocates a zero-filled region in a task's address space.

Before a memory object can be mapped, a port naming it must be acquired from the memory manager serving it.

The kernel rounds the starting address up to the next page boundary. Note that this is different from **vm_allocate**, in which the starting address is rounded down to the previous page boundary.

CAUTIONS

Do not attempt to map a memory object unless it has been provided by a memory manager that implements the memory object interface. If another type of port is specified, a thread that accesses the mapped virtual memory may become permanently hung or may receive a memory exception.

RETURN VALUE

KERN_SUCCESS
The memory object has been mapped.

KERN_NO_SPACE
There is not enough space in the task's address space to allocate the new region for the memory object.

KERN_INVALID_ARGUMENT
An illegal argument was specified.

RELATED INFORMATION

Functions: **memory_object_init**, et al., **vm_allocate**.

vm_pageable

vm_pageable Function — Specifies the pageability of a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_pageable** *(*vm_task_t *target_task,* vm_address_t *address,*
vm_size_t *size,* vm_prot_t *wired_access)*;

DESCRIPTION

The **vm_pageable** function sets the pageability privileges for a region within the specified task's address space. *wired_access* specifies an access attribute which is interpreted to specify whether the region can be paged.

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory affected may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

This call can only be made upon memory maps that allow wiring. Currently, this is only allowed for the bootstrap task and the default memory manager.

PARAMETERS

target_task
The task whose address space contains the region.

address
The starting address for the region.

size
The number of bytes in the region.

wired_access
The pageability of the region. Valid values are:

VM_PROT_DEFAULT
Wire (prevent from being paged) the region of memory.

VM_PROT_NONE
Un-wire (allow to be paged) the region of memory.

RETURN VALUE

KERN_SUCCESS
: The new pageability has been set for the region.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region.

KERN_INVALID_ARGUMENT
: The memory map does not allow the pageability of the region to be specified.

vm_protect

vm_protect Function — Sets access privileges for a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_protect** (vm_task_t *target_task,* vm_address_t *address,*
vm_size_t *size,* boolean_t *set_maximum,*
vm_prot_t *new_protection)*;

DESCRIPTION

The **vm_protect** function sets access privileges for a region within the specified task's address space. *new_protection* specifies a combination of read, write, and execute accesses that are allowed (rather than prohibited).

The region starts at the beginning of the virtual page containing *address*; it ends at the end of the virtual page containing *address* + *size* - 1. Because of this rounding to virtual page boundaries, the amount of memory protected may be greater than *size*. Use **vm_statistics** to find the current virtual page size.

The enforcement of virtual memory protection is machine-dependent. Nominally read access requires VM_PROT_READ permission, write access requires VM_PROT_WRITE permission, and execute access requires VM_PROT_EXECUTE permission. However, some combinations of access rights may not be supported. In particular, the kernel interface allows write access to require VM_PROT_READ and VM_PROT_WRITE permission and execute access to require VM_PROT_READ permission.

PARAMETERS

target_task
: The task whose address space contains the region.

address
: The starting address for the region.

size
: The number of bytes in the region.

set_maximum
: Maximum/current indicator. If true, the new protection sets the maximum protection for the region. If false, the new protection sets the current protection for the region. If the maximum protection is set be-

low the current protection, the current protection is reset to the new maximum.

new_protection
The new protection for the region. Valid values are obtained by or'-ing together the following values:

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

RETURN VALUE

KERN_SUCCESS
The new protection has been set for the region.

KERN_PROTECTION_FAILURE
The new protection increased the current or maximum protection beyond the existing maximum protection.

KERN_INVALID_ADDRESS
The address is illegal or specifies a non-allocated region.

RELATED INFORMATION

Functions: **vm_inherit**, **vm_region**.

vm_read

vm_read Function — Reads a task's virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_read** (vm_task_t *target_task,* vm_address_t *address,*
vm_size_t *size,* pointer_t **data,*
mach_msg_type_number_t **data_count)*;

DESCRIPTION

The **vm_read** function reads a portion of a task's virtual memory. It allows one task to read another task's memory.

PARAMETERS

target_task
The task whose memory is to be read.

address
The address at which to start the read.

size
The number of bytes to read.

data
[out array] The array of data returned by the read.

data_count
[out] The number of bytes in the returned array. The count converts to an integral number of pages.

NOTES

To hold the returned data array, the kernel allocates *data* as a new region in the calling task's address space, as if via **vm_allocate**. When finished with the data, use **vm_deallocate** to de-allocate this region.

RETURN VALUE

KERN_SUCCESS
The memory has been read.

KERN_INVALID_ARGUMENT
Either the address does not start on a page boundary or the size does not convert to an integral number of pages.

KERN_NO_SPACE
: There is not enough room in the calling task's address space to allocate the region for the returned data.

KERN_PROTECTION_FAILURE
: The specified region in the target task is protected against reading.

KERN_INVALID_ADDRESS
: The address is illegal or specifies a non-allocated region, or there are less than *size* bytes of data following the address.

RELATED INFORMATION

Functions: **vm_copy**, **vm_deallocate**, **vm_write**.

vm_region

vm_region Function — Returns information on a region of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_region** (vm_task_t *target_task,* vm_address_t **address,*
vm_size_t **size,* vm_prot_t **protection,*
vm_prot_t **max_protection,* vm_inherit_t
**inheritance,* boolean_t **shared,*
memory_object_name_t **object_name,*
vm_offset_t **offset)*;

DESCRIPTION

The **vm_region** function returns information on a region within the specified task's address space.

The function begins looking at *address* and continues until it finds an allocated region. If the input address is within a region, the function uses the start of that region. The starting address for the located region is returned in *address*.

PARAMETERS

target_task
The task whose address space contains the region.

address
[in/out] The address at which to start looking for a region. The function returns the starting address actually used.

size
[out] The number of bytes in the located region. The number converts to an integral number of virtual pages.

protection
[out] The current protection for the region.

max_protection
[out] The maximum protection allowed for the region.

inheritance
[out] The inheritance attribute for the region.

shared
[out] Shared indicator. If true, the region is shared by another task. If false, the region is not shared.

object_name

[out] The name of the name port for the memory object associated with the region. See **memory_object_init**.

offset

[out] The region's offset into the memory object. The region begins at this offset.

RETURN VALUE

KERN_SUCCESS

A region has been located and its information returned.

KERN_NO_SPACE

There is no region at or beyond the specified starting address.

RELATED INFORMATION

Functions: **vm_allocate**, **vm_deallocate**, **vm_inherit**, **vm_protect, memory_object_init**, et al.

vm_statistics

vm_statistics Function — Returns statistics on the kernel's use of virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_statistics** (vm_task_t *target_task,* vm_statistics_data_t **vm_stats)*;

DESCRIPTION

The **vm_statistics** function returns statistics on the kernel's use of virtual memory from the time the kernel was booted.

See **vm_statistics** for a description of the structure used.

For related information for a specific task, use **task_info**.

PARAMETERS

target_task
: The task that is requesting the statistics.

vm_stats
: [out] The structure in which the statistics will be returned.

RETURN VALUE

KERN_SUCCESS
: The statistics have been returned.

KERN_INVALID_HOST
: The host is null.

KERN_RESOURCE_SHORTAGE
: The kernel could not allocate sufficient memory.

RELATED INFORMATION

Functions: **task_info**.

Data Structures: **vm_statistics**.

vm_write

vm_write Function — Writes data to a task's virtual memory

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_write** (vm_task_t *target_task,* vm_address_t *address,*
pointer_t *data,* mach_msg_type_number_t
data_count);

DESCRIPTION

The **vm_write** function writes an array of data to a task's virtual memory. It allows one task to write to another task's memory.

Use **vm_statistics** to find the current virtual page size.

PARAMETERS

target_task
: The task whose memory is to be written.

address
: The address at which to start the write. The starting address must be on a page boundary.

data
: An array of data to be written.

data_count
: The number of bytes in the array. The size of the array must convert to an integral number of pages.

RETURN VALUE

KERN_SUCCESS
: The memory has been written.

KERN_INVALID_ARGUMENT
: Either the address does not start on a page boundary or *data_count* does not convert to an integral number of pages.

KERN_PROTECTION_FAILURE
: The specified region in the target task is protected against writing.

KERN_INVALID_ADDRESS
The address is illegal or specifies a non-allocated region, or there are less than *data_count* bytes available following the address.

RELATED INFORMATION

Functions: **vm_copy**, **vm_protect**, **vm_read**, **vm_statistics**.

CHAPTER 5

External Pager Interface Specifications

This chapter discusses the specifics of the kernel's external memory management interfaces. Interfaces that relate to the basic use of virtual memory for a task appear in the previous chapter.

default_pager_object_create

default_pager_object_create Function — Create a memory object managed by the default pager

LIBRARY

Generated by **default_pager_object.defs**.

SYNOPSIS

kern_return_t **default_pager_object_create**(memory_object_t *pager,*
memory_object_t **memory_object,* vm_size_t
object_size);

DESCRIPTION

The **default_pager_object_create** function returns an object, backed by the default pager, which is suitable for use with **vm_map**. This memory object has the same properties as does a memory object provided by **vm_allocate**: its initial contents are zero and the backing contents are temporary in that they do not persist after the memory object is destroyed. The memory object is suitable for use as non-permanent shared memory.

The default memory manager port can be obtained by calling **vm_set_default_memory_manager** with the privileged host port, specifying the “new” pager port as MACH_PORT_NULL.

PARAMETERS

pager
: A port to the default memory manager.

memory_object
: [out] The abstract memory object port for the memory object.

object_size
: The maximum size for the memory object.

RETURN VALUE

KERN_SUCCESS
: Memory object created.

RELATED INFORMATION

Functions: **memory_object_default_server**, **vm_map**, **vm_set_default_memory_manager**.

memory_object_copy

memory_object_copy Server Interface — Indicates that a memory object has been copied

SYNOPSIS

kern_return_t **memory_object_copy**(memory_object_t *old_memory_object,*
memory_object_control_t *old_memory_control,*
vm_offset_t *offset,* vm_size_t *length,*
memory_object_t *new_memory_object)*;

DESCRIPTION

A **memory_object_copy** function is called as the result of a message from the kernel indicating that the kernel has copied the specified region within the old memory object.

This call includes only the new abstract memory object port itself. The kernel will subsequently issue a **memory_object_init** call on the new abstract memory object after it has prepared the currently cached pages of the old object. When the memory manager receives the **memory_object_init** call, it is expected to reply with the **memory_object_set_attributes** call and set the ready attribute. The kernel uses the new abstract memory object, memory cache control, and memory cache name ports to refer to the new copy.

The kernel makes the **memory_object_copy** call only if:

- The memory manager had previously set the old object's copy strategy attribute to MEMORY_OBJECT_COPY_CALL (using **memory_object_set_attributes**).
- A user of the old object has asked the kernel to copy it.

Cached pages from the old memory object at the time of the copy are handled as follows:

- Readable pages may be copied to the new object without notification and with all access permissions.
- Pages not copied are locked to prevent write access.

The memory manager should treat the new memory object as temporary. In other words, the memory manager should not change the new object's contents or allow it to be mapped in another client. The memory manager can use the **memory_object_data_unavailable** call to indicate that the appropriate pages of the old object can be used to fulfill a data request.

PARAMETERS

old_memory_object
The port that represents the old (copied from) abstract memory object.

old_memory_control
The kernel memory cache control port for the old memory object.

offset
The offset within the old memory object.

length
The number of bytes copied, starting at *offset*. The number converts to an integral number of virtual pages.

new_memory_object
The new abstract memory object created by the kernel. The kernel provides all port rights (including the receive right) for the new memory object.

NOTES

It is possible for a memory manager to receive a **memory_object_data_write** request for a page of the new memory object before receiving any other requests for that data.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_unavailable**, **memory_object_init**, **memory_object_set_attributes**.

memory_object_create

memory_object_create Server Interface — Requests transfer of responsibility for a kernel-created memory object

SYNOPSIS

kern_return_t **memory_object_create**(memory_object_t *old_memory_object,* memory_object_t *new_memory_object,* vm_size_t *new_object_size,* memory_object_control_t *new_control,* memory_object_name_t *new_name,* vm_size_t *new_page_size)*;

DESCRIPTION

A **memory_object_create** function call is made as the result of a message from the kernel requesting that the default memory manager accept responsibility for the new memory object created by the kernel. The kernel makes this call only to the system default memory manager.

The new memory object initially consists of zero-filled pages. Only memory pages that are actually written are provided to the memory manager. When processing **memory_object_data_request** calls from the kernel, the default memory manager must use **memory_object_data_unavailable** for any pages that have not been written previously.

The kernel does not expect a reply to this call. The kernel assumes that the default memory manager will be ready to handle data requests to this object and does not need the confirmation of a **memory_object_set_attributes** call.

PARAMETERS

old_memory_object
: An existing abstract memory object provided by the default memory manager.

new_memory_object
: The port representing the new abstract memory object created by the kernel. The kernel provides all port rights (including the receive right) for the new memory object.

new_object_size
: The maximum size for the new object, in bytes.

new_control
: The memory cache port to be used by the memory manager when making cache management requests for the new object.

new_name
: The memory cache name port used by the kernel to refer to the new memory object data in response to **vm_region** calls.

new_page_size
The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

The kernel requires memory objects to provide temporary backing storage for zero-filled memory created by **vm_allocate** calls, issued by both user tasks and the kernel itself. The kernel allocates an abstract memory object port to represent the temporary backing storage and uses **memory_object_create** to pass the new memory object to the default memory manager, which provides the storage.

The default memory manager is a trusted system component that is identified to the kernel at system initialization time. The default memory manager can also be changed at run time using the **vm_set_default_memory_manager** call.

The contents of a kernel-created (as opposed to a user-created) memory object can be modified only in main memory. The default memory manager must not change the contents of a temporary memory object, or allow unrelated tasks to access the memory object, control, or name port.

The kernel can provide the maximum size of a temporary memory object because the object cannot be mapped by another user task.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_initialize**, **memory_object_data_unavailable**.

memory_object_data_error

memory_object_data_error Function — Indicates no data for a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_data_error**(memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *size,* kern_return_t *reason)*;

DESCRIPTION

The **memory_object_data_error** function indicates that the memory manager cannot provide the kernel with the data requested for the given region, specifying a reason for the error.

When the kernel issues a **memory_object_data_request** call, the memory manager can respond with a **memory_object_data_error** call to indicate that the page cannot be retrieved, and that a memory failure exception should be raised in any client threads that are waiting for the page. Clients are permitted to catch these exceptions and retry their page faults. As a result, this call can be used to report transient errors as well as permanent ones. A memory manager can use this call for both hardware errors (for example, disk failures) and software errors (for example, accessing data that does not exist or is protected).

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

offset
: The offset within the memory object, in bytes.

size
: The number of bytes of data (starting at *offset*). The number must convert to an integral number of memory object pages.

reason
: Reason for the error. The value could be a UNIX error code for a hardware error.

NOTES

The *reason* code is currently ignored by the kernel.

RETURN VALUE

KERN_SUCCESS
: Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_provided**, **memory_object_data_request**.

memory_object_data_initialize

memory_object_data_initialize Server Interface — Writes initial data back to a temporary memory object

SYNOPSIS

kern_return_t **memory_object_data_initialize**(memory_object_t *memory_object,* memory_object_control_t *memory_control,* vm_offset_t *offset,* pointer_t *data,* mach_msg_type_number_t *data_count)*;

DESCRIPTION

A **memory_object_data_initialize** function is called as the result of a kernel message providing the default memory manager with initial data for a kernel-created memory object. If the memory manager already has supplied data (by a previous **memory_object_data_initialize** or **memory_object_-data_write**), it should ignore this call. Otherwise, the call behaves the same as the **memory_object_data_write** call.

The kernel makes this call only to the default memory manager and only on temporary memory objects that it has created with **memory_object_create**. Note that the kernel does not make this call on objects created via **memory_-object_copy**.

PARAMETERS

memory_object
: The abstract memory object port that represents the memory object data, as supplied by the kernel in a **memory_object_create** call.

memory_control
: The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
: The offset within the memory object.

data
: The data that has been modified while cached in physical memory.

data_count
: The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_create**, **memory_object_data_write**.

memory_object_data_provided

memory_object_data_provided Function — Supplies data for a region of a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_data_provided**(memory_object_control_t *memory_control,* vm_offset_t *offset,* pointer_t *data,* mach_msg_type_number_t *data_count,* vm_prot_t *lock_value)*;

DESCRIPTION

The **memory_object_data_provided** function supplies the kernel with a range of data for the specified memory object. A memory manager normally provides data only in response to a **memory_object_data_request** call from the kernel.

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

offset
: The offset within the memory object, in bytes.

data
: The address of the data being provided to the kernel.

data_count
: The amount of data to be provided. The number must be an integral number of memory object pages.

lock_value
: One or more forms of access **not** permitted for the specified data. Valid values are:

VM_PROT_NONE
: Prohibits no access (that is, all forms of access are permitted).

VM_PROT_READ
: Prohibits read access.

VM_PROT_WRITE
Prohibits write access.

VM_PROT_EXECUTE
Prohibits execute access.

VM_PROT_ALL
Prohibits all forms of access.

NOTES

The kernel accepts only integral numbers of pages. It discards any partial pages without notification.

CAUTIONS

A memory manager must be careful when providing data that has not been explicitly requested. In particular, a memory manager must ensure that it does not provide writable data again before it receives back modifications from the kernel. This may require that the memory manager remember which pages it has provided, or that it exercise other cache control functions (via **memory_object_lock_request**) before proceeding. Currently, the kernel prohibits the overwriting of live data pages.

RETURN VALUE

KERN_SUCCESS
Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_request**, **memory_object_lock_request**.

memory_object_data_request

memory_object_data_request Server Interface — Requests data from a memory object

SYNOPSIS

kern_return_t **memory_object_data_request**(memory_object_t *memory_object,* memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *length,* vm_prot_t *desired_access)*;

DESCRIPTION

A **memory_object_data_request** function is called as the result of a kernel message requesting data from the specified memory object, for at least the access specified.

The kernel issues this call after a cache miss (that is, a page fault for which the kernel does not have the data). The kernel requests only amounts of data that are multiples of the page size included in the **memory_object_init** call.

The memory manager is expected to use **memory_object_data_provided** to return at least the specified data, with as much access as it can allow. If the memory manager cannot provide the data (for example, because of a hardware error), it can use the **memory_object_data_error** call. The memory manager can also use **memory_object_data_unavailable** to tell the kernel to supply zero-filled memory for the region.

PARAMETERS

memory_object
: The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
: The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
: The offset within the memory object.

length
: The number of bytes requested, starting at *offset*. The number converts to an integral number of virtual pages.

desired_access
: The memory access modes to be allowed for the cached data. Possible values are obtained by or'ing together the following values:

VM_PROT_READ
Allows read access.

VM_PROT_WRITE
Allows write access.

VM_PROT_EXECUTE
Allows execute access.

RETURN VALUE

KERN_SUCCESS
This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_data_error**, **memory_object_data_provided**, **memory_object_data_unavailable**.

memory_object_data_unavailable

memory_object_data_unavailable Function — Indicates no data for a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_data_unavailable**(memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *size)*;

DESCRIPTION

The **memory_object_data_unavailable** function indicates that the memory manager cannot provide the kernel with the data requested for the given region. Instead, the kernel should provide the data for this region.

A memory manager can use this call in any of the following situations:

- When the object was created by the kernel (via **memory_object_create**) and the kernel has not yet provided data for the region (via either **memory_object_data_initialize** or **memory_object_data_write**). In this case, the object is a temporary memory object; the memory manager is the default memory manager; and the kernel should provide zero-filled pages for the object.
- When the object was created by a **memory_object_copy**. In this case, the kernel should copy the region from the original memory object.
- When the object is a normal user-created memory object. In this case, the kernel should provide unlocked zero-filled pages for the region.

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** or a **memory_object_create** call.

offset
: The offset within the memory object, in bytes.

size
: The number of bytes of data (starting at *offset*). The number must convert to an integral number of memory object pages.

RETURN VALUE

KERN_SUCCESS

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_copy**, **memory_object_create**, **memory_object_data_error**, **memory_object_data_request**.

memory_object_data_unlock

memory_object_data_unlock Server Interface — Requests access to a memory object

SYNOPSIS

kern_return_t **memory_object_data_unlock**(memory_object_t *memory_object,* memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *length,* vm_prot_t *desired_access)*;

DESCRIPTION

A **memory_object_data_unlock** function is called as the result of a kernel message requesting the memory manager to permit at least the desired access to the specified data cached by the kernel. The memory manager is expected to use the **memory_object_lock_request** call in response.

PARAMETERS

memory_object
: The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
: The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
: The offset within the memory object.

length
: The number of bytes to which the access applies, starting at *offset*. The number converts to an integral number of memory object pages.

desired_access
: The memory access modes requested for the cached data. Possible values are obtained by or'ing together the following values:

VM_PROT_READ
: Allows read access.

VM_PROT_WRITE
: Allows write access.

VM_PROT_EXECUTE
: Allows execute access.

RETURN VALUE

KERN_SUCCESS

This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_lock_completed**, **memory_object_lock_-request**.

memory_object_data_write

memory_object_data_write Server Interface — Writes changed data back to a memory object

SYNOPSIS

kern_return_t **memory_object_data_write**(memory_object_t *memory_object,* memory_object_control_t *memory_control,* vm_offset_t *offset,* pointer_t *data,* mach_msg_type_number_t *data_count)*;

DESCRIPTION

A **memory_object_data_write** function is called as the result of a kernel message providing the memory manager with data that has been modified while cached in physical memory.

The kernel writes back only data that has been modified. When the memory manager no longer needs the data (for example, after the data has been written to permanent storage), it should use **vm_deallocate** to release the memory resources.

PARAMETERS

memory_object
: The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
: The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
: The offset within the memory object.

data
: The data that has been modified while cached in physical memory.

data_count
: The number of bytes to be written, starting at *offset*. The number converts to an integral number of memory object pages.

NOTES

The kernel can flush clean (that is, un-modified) pages at its own discretion. As a result, the memory manager cannot rely on the kernel to keep a copy of its data or even to provide notification that its data has been discarded.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **vm_deallocate**.

memory_object_default_server

memory_object_default_server Function — Handles kernel messages for the default memory manager

LIBRARY

Generated by **memory_object_default.defs**.

SYNOPSIS

boolean_t **memory_object_default_server**(mach_msg_header_t **in_msg,*
mach_msg_header_t **out_msg)*;

DESCRIPTION

The **memory_object_default_server** function is the MIG generated server handling function to handle messages from the kernel targeted to the default memory manager. This server function only handles messages unique to the default memory manager. Messages that are common to all memory managers are handled by **memory_object_server**.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_default_server** function performs all necessary argument handling for a kernel message and calls one of the following functions to interpret the message:

- **memory_object_create**
- **memory_object_data_initialize**

PARAMETERS

in_msg
The memory manager message received from the kernel.

out_msg
A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_server**, **memory_object_create**, **memory_object_data_initialize**.

memory_object_destroy

memory_object_destroy Function — Shuts down a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_destroy**(memory_object_control_t
memory_control, kern_return_t *reason)*;

DESCRIPTION

The **memory_object_destroy** function tells the kernel to shut down the specified memory object. As a result of this call, the kernel no longer supports paging activity or any memory object calls on the memory object. The kernel issues a **memory_object_terminate** call to pass to the memory manager all rights to the memory object port, the memory control port, and the memory name port.

To ensure that any modified cached data is returned before the object is terminated, the memory manager should call **memory_object_lock_request** with *should_flush* set and a lock value of VM_PROT_WRITE before it makes the **memory_object_destroy** call.

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

reason
: An error code indicating when the object must be destroyed.

NOTES

The *reason* code is currently ignored by the kernel.

RETURN VALUE

KERN_SUCCESS
: Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_lock_request**, **memory_object_terminate**.

memory_object_get_attributes

memory_object_get_attributes Function — Returns current attributes for a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_get_attributes**(memory_object_control_t *memory_control,* boolean_t **object_ready,* boolean_t **may_cache_object,* memory_object_copy_strategy_t **copy_strategy)*;

DESCRIPTION

The **memory_object_get_attributes** function retrieves the current attributes for the specified memory object.

PARAMETERS

memory_control
The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

object_ready
[out] Ready indicator. If true, the kernel can issue new data and unlock requests on the memory object.

may_cache_object
[out] Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy
[out] How the kernel should handle copying of regions associated with the memory object. Possible values are:

MEMORY_OBJECT_COPY_NONE
Use normal procedure when copying the memory object's data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.

MEMORY_OBJECT_COPY_CALL
Notify the memory manager (via **memory_object_copy**) before copying any data.

MEMORY_OBJECT_COPY_DELAY
: Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

RETURN VALUE

KERN_SUCCESS
: Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_copy**, **memory_object_set_attributes**.

memory_object_init

memory_object_init Server Interface — Initializes a memory object

SYNOPSIS

kern_return_t **memory_object_init**(memory_object_t *memory_object,*
memory_object_control_t *memory_control,*
memory_object_name_t *memory_object_name,*
vm_size_t *memory_object_page_size)*;

DESCRIPTION

A **memory_object_init** function is called as the result of a kernel message notifying a memory manager that the kernel has been asked to map the specified memory object into a task's virtual address space.

When asked to map a memory object for the first time, the kernel responds by making a **memory_object_init** call on the abstract memory object. This call is provided as a convenience to the memory manager, to allow it to initialize data structures and prepare to receive other requests.

In addition to the abstract memory object port itself, the call provides the following two ports:

- A memory cache control port that the memory manager can use to control use of its data by the kernel. The memory manager gets send rights for this port.
- A memory cache name port that the kernel will use to identify the memory object to other tasks.

The kernel holds send rights for the abstract memory object port, and both send and receive rights for the memory cache control and name ports.

The call also supplies the virtual page size to be used for the memory mapping. The memory manager can use this size to detect mappings that use different data structures at initialization time, or to allocate buffers for use in reading data.

If a memory object is mapped into the address space of more than one task on different hosts (with independent kernels), the memory manager will receive a **memory_object_init** call from each kernel, containing a unique set of control and name ports. Note that each kernel may also use a different page size.

PARAMETERS

memory_object
The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
: The memory cache control port to be used by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

memory_object_name
: The memory cache name port used by the kernel to refer to the memory object data in response to **vm_region** calls.

memory_object_page_size
: The page size used by the kernel. All calls involving this kernel must use data sizes that are integral multiples of this page size.

NOTES

When the memory manager is ready to accept data requests for this memory object, it must call **memory_object_set_attributes** with the *object_ready* attribute set. Otherwise, the kernel will not process requests on this object.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_set_attributes**, **memory_object_terminate**.

memory_object_lock_completed

memory_object_lock_completed Server Interface — Indicates completion of a consistency control call

SYNOPSIS

kern_return_t **memory_object_lock_completed**(memory_object_t *memory_object,* memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *length)*;

DESCRIPTION

A **memory_object_lock_completed** function is called as the result of a kernel message confirming the kernel's action in response to a **memory_object_-lock_request** call from the memory manager. The memory manager can use the **memory_object_lock_request** call to:

- Alter access restrictions specified in the **memory_object_data_provided** call or a previous **memory_object_lock_request** call.
- Write back modifications made in memory.
- Invalidate its cached data.

When the kernel completes the requested actions, it calls **memory_object_-lock_completed** (asynchronously) using the port explicitly provided in the **memory_object_lock_request** call. Because the memory manager cannot know which pages have been modified, or even which pages remain in the cache, it cannot know how many pages will be written back in response to a **memory_object_lock_request** call. Receiving the **memory_object_lock_-completed** call is the only sure means of detecting completion. The completion call includes the offset and length values from the consistency request to distinguish it from other consistency requests.

PARAMETERS

memory_object
: The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
: The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

offset
: The offset within the memory object.

length

The number of bytes to which the call refers, starting at *offset*. The number converts to an integral number of memory object pages.

RETURN VALUE

KERN_SUCCESS

This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_lock_request**.

memory_object_lock_request

memory_object_lock_request Function — Restricts access to memory object data

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_lock_request**(memory_object_control_t *memory_control,* vm_offset_t *offset,* vm_size_t *size,* boolean_t *should_clean,* boolean_t *should_flush,* vm_prot_t *lock_value,* mach_port_t *reply_to)*;

DESCRIPTION

The **memory_object_lock_request** function allows the memory manager to make the following requests of the kernel:

- Clean the pages within the specified range by writing back all changed (that is, dirty) pages. The kernel uses the **memory_object_data_write** call to write back the data. The *should_clean* parameter must be set to true.
- Flush all cached data within the specified range. The kernel invalidates the range of data and revokes all uses of that data. The *should_flush* parameter must be set to true.
- Alter access restrictions specified in the **memory_object_data_provided** call or a previous **memory_object_lock_request** call. The *lock_value* parameter must specify the new access restrictions. Note that this parameter can be used to unlock previously locked data.

Once the kernel performs all of the actions requested by this call, it issues a **memory_object_lock_completed** call using the *reply_to* port.

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

offset
: The offset within the memory object, in bytes.

size
: The number of bytes of data (starting at *offset*) to be affected. The number must convert to an integral number of memory object pages.

should_clean

Clean indicator. If true, the kernel writes all modified pages back to the memory manager.

should_flush

Flush indicator. If true, the kernel flushes all pages within the range.

lock_value

One or more forms of access **not** permitted for the specified data. Valid values are:

VM_PROT_NONE

Prohibits no access (that is, all forms of access are permitted).

VM_PROT_READ

Prohibits read access.

VM_PROT_WRITE

Prohibits write access.

VM_PROT_EXECUTE

Prohibits execute access.

VM_PROT_ALL

Prohibits all forms of access.

reply_to

The response port to be used by the kernel on a call to **memory_object_lock_completed**, or MACH_PORT_NULL if no response is required.

NOTES

The **memory_object_lock_request** call affects only data that is cached at the time of the call. Access restrictions cannot be applied to pages for which data has not been provided.

When a running thread requires an access that is currently prohibited, the kernel issues a **memory_object_data_unlock** call specifying the access required. The memory manager can then use **memory_object_lock_request** to relax its access restrictions on the data.

To indicate that an unlock request is invalid (that is, requires permission that can never be granted), the memory manager must first flush the page. When the kernel requests the data again with the higher permission, the memory manager can indicate the error by responding with a call to **memory_object_data_error**.

RETURN VALUE

KERN_SUCCESS

Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_data_provided**, **memory_object_data_unlock**, **memory_object_lock_completed**.

memory_object_server

memory_object_server Function — Handles kernel messages for a memory manager

LIBRARY

Generated by **memory_object.defs**.

SYNOPSIS

boolean_t **memory_object_server**(mach_msg_header_t **in_msg,*
mach_msg_header_t **out_msg)*;

DESCRIPTION

The **memory_object_server** function is the MIG generated server handling function to handle messages from the kernel targeted to a memory manager.

A *memory manager* is a server task that responds to specific messages from the kernel in order to handle memory management functions for the kernel. The **memory_object_server** function performs all necessary argument handling for a kernel message and calls one of the following functions to interpret the message:

- **memory_object_init**
- **memory_object_data_write**
- **memory_object_data_request**
- **memory_object_data_unlock**
- **memory_object_lock_completed**
- **memory_object_copy**
- **memory_object_terminate**

PARAMETERS

in_msg
The memory manager message received from the kernel.

out_msg
A reply message. No messages to a memory manager expect a direct reply, so this field is not used.

RETURN VALUE

TRUE
The message was handled and the appropriate function was called.

FALSE
The message did not apply to this memory management interface and no other action was taken.

RELATED INFORMATION

Functions: **memory_object_default_server**, **memory_object_copy**, **memory_object_data_request**, **memory_object_data_unlock**, **memory_object_data_write**, **memory_object_init**, **memory_object_lock_completed**, **memory_object_terminate**.

memory_object_set_attributes

memory_object_set_attributes Function — Sets attributes for a memory object

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **memory_object_set_attributes**(memory_object_control_t
memory_control, boolean_t *object_ready,*
boolean_t *may_cache_object,*
memory_object_copy_strategy_t *copy_strategy)*;

DESCRIPTION

The **memory_object_set_attributes** function allows the memory manager to set performance-related attributes for the specified memory object. These attributes control whether the kernel is permitted to:

- Make data or unlock requests on behalf of clients (*object_ready* parameter).
- Retain data from a memory object even after all address space mappings have been de-allocated (*may_cache_object* parameter).
- Perform optimizations for virtual memory copy operations (*copy_strategy* parameter).

PARAMETERS

memory_control
: The memory cache control port to be used by the memory manager for cache management requests. This port is provided by the kernel in a **memory_object_init** call.

object_ready
: Ready indicator. If true, the kernel can issue new data and unlock requests on the memory object.

may_cache_object
: Cache indicator. If true, the kernel can cache data associated with the memory object, even if virtual memory references to it are removed.

copy_strategy
: How the kernel should handle copying of regions associated with the memory object. Valid values are:

MEMORY_OBJECT_COPY_NONE
: Use normal procedure when copying the memory object's data. Normally, the kernel requests each page with read access, copies the data, and then (optionally) flushes the data.

MEMORY_OBJECT_COPY_CALL
: Notify the memory manager (via **memory_object_copy**) before copying any data.

MEMORY_OBJECT_COPY_DELAY
: Use copy-on-write technique. This strategy allows the kernel to efficiently copy large amounts of data and guarantees that the memory manager will not externally modify the data. It is the most commonly used copy strategy.

NOTES

Changing a memory object to be not ready does not affect data and unlock requests already in progress. Such requests will not be aborted or reissued.

Sharing cached data among all the clients of a memory object can have a major impact on performance, especially if it can be extended across successive, as well as concurrent, uses. For example, the memory objects that represent program images can be used regularly by different programs. By retaining the data for these memory objects in cache, the number of secondary storage accesses can be reduced significantly.

RETURN VALUE

KERN_SUCCESS
: Since this function does not receive a reply message, it has no return value. Only message transmission errors apply.

RELATED INFORMATION

Functions: **memory_object_copy**, **memory_object_get_attributes**, **memory_object_init**.

memory_object_terminate

memory_object_terminate Server Interface — Relinquishes access to a memory object

SYNOPSIS

kern_return_t **memory_object_terminate**(memory_object_t *memory_object,*
memory_object_control_t *memory_control,*
memory_object_name_t *memory_object_name)*;

DESCRIPTION

A **memory_object_terminate** function is called as the result of a kernel message notifying a memory manager that no mappings of the specified memory object remain. The kernel makes this call to allow the memory manager to clean up data structures associated with the de-allocated mappings. The call provides receive rights to the memory cache control and name ports so that the memory manager can destroy the ports (via **mach_port_deallocate**). The kernel also relinquishes its send rights for all three ports.

The kernel terminates a memory object only after all address space mappings of the object have been de-allocated, or upon explicit request by the memory manager.

PARAMETERS

memory_object
The abstract memory object port that represents the memory object data, as supplied to the kernel in a **vm_map** call.

memory_control
The memory cache control port to be used for a response by the memory manager. If the memory object has been supplied to more than one kernel, this parameter identifies the kernel that is making the call.

memory_object_name
The memory cache name port used by the kernel to refer to the memory object data in response to **vm_region** calls.

NOTES

If a client thread calls **vm_map** to map a memory object while the kernel is calling **memory_object_terminate** for the same memory object, the **memory_object_init** call may appear before the **memory_object_terminate** call. This sequence is indistinguishable from the case where another kernel is issuing a **memory_object_init** call. In other words, the control and name ports included in the initialization will be different from those included in the termination. A memory manager must be aware that this sequence can occur even when all mappings of a memory object take place on the same host.

RETURN VALUE

KERN_SUCCESS
: This value is ignored since the call is made by the kernel, which does not wait for a reply.

RELATED INFORMATION

Functions: **memory_object_destroy**, **memory_object_init**, **mach_-port_deallocate**.

vm_set_default_memory_manager

vm_set_default_memory_manager Function — Sets the default memory manager.

LIBRARY

libmach.a

#include <mach.h> (**<mach/mach_interface.h>**)

SYNOPSIS

kern_return_t **vm_set_default_memory_manager**(host_priv_t *host,*
mach_port_t **default_manager)*;

DESCRIPTION

The **vm_set_default_memory_manager** function establishes the default memory manager for a host.

PARAMETERS

host
: The control port naming the host for which the default memory manager is to be set.

default_manager
: [in/out] A memory manager port to the new default memory manager. If this value is MACH_PORT_NULL, the old memory manager is not changed. The old memory manager port is returned in this variable.

RETURN VALUE

KERN_SUCCESS
: The old default memory port was returned and the new manager established.

KERN_INVALID_ARGUMENT
: The supplied host port is not the privileged host port.

RELATED INFORMATION

Functions: **memory_object_create**, **vm_allocate**.