MACH Kernel Interface Manual

Robert V. Baron
David Black
William Bolosky
Jonathan Chew
Richard P. Draves
David B. Golub
Richard F. Rashid
Avadis Tevanian, Jr.
Michael Wayne Young

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Version of:
23 August 1990

Abstract

MACH is an operating system kernel under development at Carnegie-Mellon University to support distributed and parallel computation. MACH is designed to support computing environments consisting of networks of uniprocessors and multiprocessors. This manual describes the interface to the MACH kernel in detail. The MACH system currently runs on a wide variety of uniprocessor and multiprocessor architectures.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-84-C-0467.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.
1. Introduction

MACH is a communication-oriented operating system kernel providing:

- multiple tasks, each with a large, paged virtual memory space,
- multiple threads of execution within each task, with a flexible scheduling facility,
- flexible sharing of memory between tasks,
- message-based interprocess communication,
- transparent network extensibility, and
- a flexible capability-based approach to security and protection.

MACH supports multiprocessor scheduling and is currently in use on both general purpose multiprocessor and uniprocessor systems. MACH is currently supported at CMU on the DEC VAX 8650, 8600, 11/785, 11/780, 11/750 and MicroVAX II, the IBM RT/PC and the SUN 3. It also will run as a shared memory multiprocessor system on the four processor VAX 11/784 and two processor VAX 11/782, the two processor VAX 8300, the VAX 8200 with one or more CPUs, the 20 processor Encore MultiMax and the 30 processor Sequent Balance 21000. Ports of MACH to other computers are in progress.

1.1. Overall system organization

As a working environment for developing application programs, MACH can be viewed as being split into two components:

- a small, extensible system kernel which provides scheduling, virtual memory and interprocess communications and
- several, possibly parallel, operating system support environments which provide the following two items: 1) distributed file access and remote execution 2) emulation for established operating system environments such as UNIX.

The extensibility of the basic MACH kernel facilitates the incorporation of new operating system functions; user-state programs can simply be added to the existing kernel without the need to modify the underlying kernel base. The basic kernel abstractions have been designed in such a way as to provide for completely transparent network extensibility of all kernel functions.

MACH is 4.3bsd UNIX binary compatible on VAX architecture machines. In addition, the MACH environment includes an internal kernel debugger, transparent network interprocess communication, remote execution facilities, a transparent remote UNIX file system and support for graphics workstations.

1.2. Basic kernel functionality

The MACH kernel supports the following basic abstractions:

- A task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space (potentially sparse) and protected access to system resources (such as processors, port capabilities and virtual memory).

- A thread is the basic unit of execution. It consists of all processor state (e.g.hardware registers) necessary for independent execution. A thread executes in the virtual memory and port rights context of a single task. The conventional notion of a process is, in MACH, represented by a task with a single thread of control.
• A **port** is a simplex communication channel -- implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in **MACH**. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports which represent them.

• A **port set** is a group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports.

• A **message** is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

• A **memory object** is a secondary storage object that is mapped into a task’s virtual memory. Memory objects are commonly files managed by a file server, but as far as the **MACH** kernel is concerned, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data.

Message-passing is the primary means of communication both among tasks, and between tasks and the operating system kernel itself. The only functions implemented by system traps are those directly concerned with message communication; all the rest are implemented by messages to a task’s task_port.

The **MACH** kernel functions can be divided into the following categories:

• basic message primitives and support facilities,
• port and port set management facilities,
• task and thread creation and management facilities,
• virtual memory management functions,
• operations on memory objects.

**MACH** and other server interfaces are defined in a high-level remote procedure call language called **MIG**; from that definition, interfaces for **C** are generated. In the future, **MIG** may generate interfaces in other languages. In this manual, calls are shown in the **C** language.

All **MACH** kernel procedures return a value indicating the success or reason for failure of that request. The errors unique to each function are described with those functions; however, since all requests involve primitive message operations, errors described in that section may also apply.

### 1.3. User operating system environments

In addition to the facilities provided directly by the kernel, **MACH** also provides for complete emulation of all 4.3bsd functions as described in the 4.3bsd manual. This emulation is completely transparent to user programs and requires no special libraries or other utilities. On all VAX hardware **MACH** is binary compatible with 4.3bsd.

This manual does not reproduce descriptions of the UNIX system calls. Programmers wishing to use the functions provided within these environments should consult the relevant UNIX system manuals.
2. Message primitives

2.1. Basic terms

MACH message primitives manipulate three distinct objects:

1. **ports** - protected kernel objects to which messages may be sent and logically queued until reception,
2. **port sets** - protected kernel objects which combine multiple port queues and from which messages may be dequeued, and
3. **messages** - ordered collections of typed data consisting of a fixed size message header and a variable size message body.

2.2. Ports

Access rights to a port consist of the ability to send to, receive from, or own that port. A task may hold just send rights or any combination of receive and ownership rights plus send rights. Threads within a task may only refer to ports to which that task has been given access. When a new port is created within a task, that task is given all three access rights to that port.

The port access rights are operationally defined as follows:

- **Send access** to a port implies that a message can be sent to that port. Should the port be destroyed during the time a task has send access, a message will be sent to that task by the kernel indicating that the port has disappeared.
- **Receive access** to a port allows a message to be dequeued from that port. Only one task may have receive access for a given port at a time; however, more than one thread within that task may concurrently attempt to receive messages from a given port. Receive access implies send rights.
- **Ownership** of a port implies that, should the task with receive access to that port relinquish its receive access, the receive access to the port will be sent to the owner task. Likewise, should ownership be relinquished, the ownership rights are sent by the kernel to the receiving task. The name ownership is somewhat misleading as all it really means is that the task is a backup receiver if the current receiver gives up its rights. As with receive access, only one task may hold ownership access to any given port. Ownership implies send rights. **NOTE**: the ownership abstraction is considered obsolete and has been replaced with the use of a backup port. This is a port associated with a primary port, to which the receive rights of the primary port will be sent in the event of an attempted destruction of the primary port. Current versions of MACH implement both mechanisms, but the ownership rights may disappear in future releases.

Port access rights can be passed in messages. They are interpreted by the kernel and transferred from the sender to the kernel upon message transmission and to the receiver upon message reception. Send rights are kept by the original task as well as being transmitted to the receiver task, but receive rights and ownership rights are removed from the original task at the time of the send, and appear in the user task when the receive is done. During the time between a send and receive, the kernel holds the rights and any messages sent to the port will be queued awaiting a new task to receive on the port. If the task that was intended to receive the rights dies before it does the receive, the rights are handled as though the receive had been done before the task died; that is receive rights are transferred to the owner
or ownership is transferred to the receiver. If the receiver and owner are both dead, the port is destroyed.

The message queue associated with a port is of finite length and thus may become full. Threads may exercise several options for handling the case of message transmission to a full queue (see msg_send below). Unless a specific option is set, msg_send will block until the message can be queued.

2.3. Port sets

Conceptually, a port set is a bag holding zero or more receive rights. A port set allows a thread to block waiting for a message sent to any of several ports. A port may be a member of at most one port set at any time.

A task's port set right, created by port_set_allocate, allows the task to receive a message from the port set with msg_receive and manipulate the port set with port_set_add, port_set_remove, port_set_status, and port_set_deallocate. Unlike port rights, a port set right may not be passed in messages.

2.4. Port names

Every task has its own port name space, used for port and port set names. For example, one task with receive and ownership rights for a port may know the port by the name 13, while another task with send rights for the same port may know it by the name 17. A task only has one name for a port, so if the task with send rights named 17 receives another message carrying send rights for the same port, the arriving rights will also be named 17.

Typically these names are small integers, but that is implementation dependent. When a task receives a message carrying rights for a new port, the MACH kernel is free to choose any unused name. The port_rename call can be used to change a task's name for a port.

2.5. Port types

There are several type definitions for ports used in this manual and defined in <mach/port.h>. The type port_name_t is used to refer to a port to which the task may have no rights. When this type is used in a message definition no port rights are sent in the message and the kernel does no mapping of ports. The type port_set_name_t is used to refer to a port set and does not imply any rights to the set. Only port set names can be passed in messages. In order to pass the rights to a port set, a task must pass each port separately and the receiving port must then define a new port set with consisting of those ports. The types port_t, port_rcv_t and port_all_t are used to imply a port to which the task has the specified rights. Typically port_t is used for a port with any rights. One of these types must be used in the message definition if ports rights are to be sent in the message. All of these types are defined to be the same basic C types, so that they can be used interchangeably in calls to primitives.

Most of the MACH calls take a task or thread as their first argument where this argument is said to be the target task/thread. In most cases the task or thread is the one doing the call. In those cases any port_name_t arguments represent ports to which the task has or receives rights. But in the case where task is not the caller, then the target task gets the rights but doesn’t know the name, and the caller gets the name but does not have any rights to the port.
2.6. Messages

A message consists of a fixed header, followed by a variable amount of data. The C type definition for the message header is as follows:

```c
typedef struct {
    int :24,
    msg_simple : 8;
    int msg_size;
    int msg_type;
    port_t msg_local_port;
    port_t msg_remote_port;
    int msg_id;
} msg_header_t;
```

The `msg_local_port` and `msg_remote_port` fields are used to name the ports on which a message is to be received or sent. In the case of `msg_receive` this may be either a port or a port set. The `msg_size` field is used to describe the size of the message to be sent, or the maximum size of the message which can be received. The size includes the header and inline data and is given in bytes. The `msg_simple` field is used to indicate that no ports or out-of-line data are contained in the body. The `msg_id` field may be used by user programs to identify the meaning of this message to the intended recipient.

The variable data part of a message consists of an array of descriptors and data. Each data descriptor is of the form:

```c
typedef struct {
    unsigned int msg_type_name : 8,
    /* What kind of data */
    msg_type_size : 8,
    /* How many bits is each item */
    msg_type_number : 12,
    /* How many items are there */
    msg_type_inline : 1,
    /* If true, actual data follows; */
    msg_type_longform : 1,
    /* Name, size, number follow */
    msg_type_deallocate : 1;
    /* Deallocate port rights or memory */
} msg_type_t;
```

`msg_type_name` describes the basic type of data comprising this object. There are several system-defined data types, including:

- Ports, including combinations of send, receive, and ownership rights,
- Port and port set names. This is the same language data type as port rights, but the message only carries a task's name for a port and doesn't cause any transferal of rights.
- Simple data types, such as integers, characters, and floating point values.

`msg_type_size` indicates the size in bits of the basic object named in the `msg_type_name` field.

`msg_type_number` indicates the number of items of the basic data type present after the type descriptor.

`msg_type_inline` indicates that the actual data is included after the type descriptor; otherwise, the word following the descriptor is a pointer to the data to be sent.
msg_type_deallocate
indicates that the port rights and/or data pointed to in this object are to be deallocated after the queueing of this message. Receive and ownership rights may not be deallocated with msg_type_deallocate.

msg_type_longform
indicates that the name, size, and number fields were too long to fit in the structure described above. Instead, the data type descriptor is described by the following structure:

```c
typedef struct  {
    msg_type_t  msg_type_header;
    short       msg_type_long_name;
    short       msg_type_long_size;
    int         msg_type_long_number;
} msg_type_long_t;
```

A data item or a pointer to data follows each data descriptor.

All the C types and constants needed to use the message functions are defined in <mach/message.h>. The declarations in this section are taken from this file.
msg_send
#include <mach/message.h>
msg_return_t msg_send(header, option, timeout)
    msg_header_t *header;
    msg_option_t option;
    msg_timeout_t timeout;

Arguments

header The address of the message to be sent. A message consists of a fixed sized header followed by a variable number of data descriptors and data items. See <mach/message.h> for a definition of the message structure.

timeout In the event that the destination port is full and the SEND_TIMEOUT option has been specified, this value specifies the maximum wait time (in milliseconds).

option The failure conditions under which msg_send should terminate; the value of this parameter is an or'ed combination of the following two options. Unless one of the two following values for the option parameter is explicitly specified, msg_send does not return until the message is successfully queued for the intended receiver.

SEND_TIMEOUT specifies that the msg_send request should terminate after the timeout period has elapsed, even if the kernel has been unable to queue the message.

SEND_NOTIFY allows the sender to give exactly one message to the operating system without being suspended should the destination port be full. When another message can be forced to the receiving port's queue using SEND_NOTIFY, the sending task receives a NOTIFY_MSG_ACCEPTED notification. A second attempt to send a message with the notify option before the notification arrives results in an error. If SEND_TIMEOUT is also specified, msg_send will wait until the specified timeout has elapsed before invoking the SEND_NOTIFY option.

SEND_INTERRUPT Specifies that msg_send should return if a software interrupt occurs in this thread.

MSG_OPTION_NONE A constant defined as zero which may be used to specify that neither of the previous options are wanted.

Description

msg_send transmits a message from the current task to the remote port specified in the message header field (msg_remote_port). The message consists of its header, followed by a variable number of data descriptors and data items. (See the introduction to this section for details on message formatting.)

If the msg_local_port field is not set to PORT_NULL, send rights to that port will be passed to the receiver of this message. The receiver task may use that port to send a reply to this message.

If the SEND_NOTIFY option is used and this call returns a SEND_WILL_NOTIFY code, then the user can expect to receive a notify message from the kernel. This message will either be a NOTIFY_MSG_ACCEPTED or a NOTIFY_PORT_DELETED message depending on what happened to the queued message. The first and only data item in these messages is the port to which the original message was sent. The ids and formats for these messages are defined in <mach/notify.h>.
Returns

SEND_SUCCESS   The message has been queued for the destination port.
SEND_INVALID_MEMORY   The message header or body was not readable by the calling task, or the message body specified out-of-line data which was not readable.
SEND_INVALID_PORT   The message refers to a name for which the current task does not have access, or to which access was explicitly removed from the current task (see port_deallocate) while waiting for the message to be posted, or a msg_type_name field in the message specifies rights that the name doesn’t denote in the task (eg, specifying MSG_TYPE_SEND and supplying a port set’s name).
SEND_TIMED_OUT   The message was not sent since the destination port was still full after timeout milliseconds.
SEND_WILL_NOTIFY   The destination port was full but the SEND_NOTIFY option was specified. A notification message will be sent when the message can be posted.
SEND_NOTIFY_IN_PROGRESS   The SEND_NOTIFY option was specified but a notification request is already outstanding for this thread and given destination port.

See Also

msg_receive, msg_rpc
msg_receive

#include <mach/message.h>
#include <mach/port.h>

msg_return_t msg_receive(header, option, timeout)
    msg_header_t *header; /* in/out */
    msg_option_t option;
    msg_timeout_t timeout;

Arguments

header The address of a buffer in which the message is to be received. Two fields of the message header must be set before the call is made: msg_local_port is set to the name of the port or port set from which the message is to be received and msg_size must be set to the maximum size of the message that may be received. It must be less than or equal to the size of the buffer.

timeout If RCV_TIMEOUT is specified this value is the maximum time in milliseconds to wait for a message before giving up.

option The failure conditions under which msg_receive should terminate; the value of this parameter is a bit or'd combination the following two options. Unless one of the two following values for the option parameter is explicitly specified, msg_receive does not return until a message has been received.

RCV_TIMEOUT Specifies that msg_receive should return when the specified timeout elapses, if a message has not arrived by that time; if not specified, the timeout will be ignored (i.e. infinite).

RCV_NO_SENDERS Specifies that msg_receive should return if the receiver and owner tasks have the only access rights to the port specified in the message header. (Not implemented yet)

RCV_INTERRUPT Specifies that msg_receive should return when a software interrupt has occurred in this thread.

MSG_OPTION_NONE Specifies that none of the above options are desired.

Description

msg_receive retrieves the next message from a port or port set specified in the msg_local_port field of the specified message header. If a port is specified, the port may not be a member of a port set. The msg_local_port field will be set to the specific port on which the message was found.

If a port set is specified, the msg_receive will retrieve messages sent to any of the set’s member ports. It is not an error for the port set to have no members, or for members to be added and removed from a port set while a msg_receive on the port set is in progress.

The message consists of its header, followed by a variable amount of data; the message header supplied to msg_receive must specify the maximum size of the message which can be received into the buffer provided. (See the introduction to this section for details on message formatting).

If no messages are present on the port(s) in question, msg_receive will wait until a message arrives, or until one of the specified termination conditions is met (see above for discussion of the option parameter).
If the received messages contains out-of-line data (i.e. for which the \texttt{msg\_type\_inline} attribute was specified as \texttt{FALSE}), the data will be returned in a newly-allocated region of memory; the message body will contain a pointer to that new region. (See \texttt{vm\_allocate} call for a description of the state of newly-allocated memory.) The user may wish to deallocate this memory when the data is no longer needed.

**Returns**

- **RCV\_SUCCESS**: The message has been received.
- **RCV\_INVALID\_MEMORY**: The message specified was not writable by the calling task.
- **RCV\_INVALID\_PORT**: An attempt was made to receive on a port to which the calling task does not have the proper access, or which was deallocated (see \texttt{port\_deallocate}) while waiting for a message.
- **RCV\_TOO\_LARGE**: The message header and body combined are larger than the size specified by \texttt{msg\_size}.
- **RCV\_NOT\_ENOUGH\_MEMORY**: The message to be received contains more out-of-line data than can be allocated in the receiving task.
- **RCV\_TIMED\_OUT**: The message was not received after \texttt{timeout} milliseconds.
- **RCV\_ONLY\_SENDER**: An attempt was made to receive on a port to which only the receive and/or owner have access, and the \texttt{RCV\_NO\_SENDERS} option was specified.
- **RCV\_INTERRUPTED**: A software interrupt occurred.
- **RCV\_PORT\_CHANGE**: The port specified was moved into a port set during the duration of the \texttt{msg\_receive} call.

**See Also**

\texttt{msg\_rpc}, \texttt{msg\_send}
msg_rpc
#include <mach/message.h>
#include <mach/port.h>

msg_return_t msg_rpc(header, option, rcv_size, 
  send_timeout, rcv_timeout)
  msg_header_t *header;  /* in/out */
  msg_option_t option;
  msg_size_t rcv_size;
  msg_timeout_t send_timeout;
  msg_timeout_t rcv_timeout;

Arguments
header
Address of a message buffer which will be used for both msg_send and msg_receive. This buffer contains a message header followed by the data for the message to be sent. The msg_remote_port field specifies the port to which the message is to be sent. The msg_local_port field specifies the port on which a message is then to be received; if this port is the special value PORT_DEFAULT, it will be replaced by the value PORT_NULL for the purposes of the msg_send operation.

option
A union of the option parameters for the component operations. (see msg_send and msg_receive)

rcv_size
The maximum size allowed for the received message; this must be less than or equal to the size of the message buffer. The msg_size field in the header specifies the size of the message to be sent.

send_timeout;rcv_timeout
The timeout values to be applied to the component operations. These are only used if the options SEND_TIMEOUT and/or RCV_TIMEOUT are specified.

Description
msg_rpc is a hybrid call which performs a msg_send followed by a msg_receive, using the same message buffer.

Returns
RPC_SUCCESS
message was successfully sent and a reply was received.

FAILURES
are the same as those for msg_send and msg_receive; any error during the msg_send portion will terminate the call.

See Also
msg_receive, msg_send
3. Port and port set primitives
port_names
#include <mach.h>

kern_return_t port_names(task,
    portnames, portnamesCnt,
    port_types, port_typesCnt)

    task_t task;
    port_name_array_t *portnames; /* out array */
    unsigned int *portnamesCnt; /* out */
    port_type_array_t *port_types; /* out array */
    unsigned int *port_typesCnt; /* out */

Arguments

    task The task whose port name space is queried.
    portnames The names of the ports and port sets in the task’s port name space, in no
        particular order.
    portnamesCnt The number of names returned.
    port_types The type of each corresponding name. Indicates what kind of right the task
        holds for the port or port set.
    port_typesCnt Should be the same as portnamesCnt.

Description

    port_names returns the currently valid ports and port set names of task. For each name, it also
    returns what type of rights task holds. portnames and port_types are arrays that are automatically
    allocated when the reply message is received. The user may wish to vm_deallocate them when the
    data is no longer needed.

Returns

    KERN_SUCCESS The call succeeded.
    KERN_INVALID_ARGUMENT task was invalid.

See Also

    port_type, port_status, port_set_status
port_type
#include <mach.h>

kern_return_t port_type(task, port_name, port_type)
    task_t task;
    port_name_t port_name;
    port_type_t *port_type; /* out */

Arguments
    task        The task whose port name space is queried.
    port_name   The name being queried.
    port_type   The type of the name. Indicates what kind of right the task holds for the port
                or port set.

Description
    port_type returns information about task’s rights for a specific name in its port name space.

Returns
    KERN_SUCCESS    The call succeeded.
    KERN_INVALID_ARGUMENT
                task was invalid or task did not have any rights named port_name.

See Also
    port_names, port_status, port_set_status
port_rename
#include <mach.h>

kern_return_t port_rename(task, old_name, new_name)
    task_t task;
    port_name_t old_name;
    port_name_t new_name;

Arguments
    task                   The task whose port name space is changed.
    old_name               The name being changed.
    new_name               The new value for old_name.

Description
    port_rename changes the name by which a port or port set is known to task. new_name must not already be in use, and it can't be a distinguished value like PORT_NULL.

Returns
    KERN_SUCCESS          The call succeeded.
    KERN_NAME_EXISTS      task already has a right named new_name.
    KERN_INVALID_ARGUMENT task was invalid or task did not have any rights named old_name or new_name was an invalid name.

See Also
    port_names
port_allocate
#include <mach.h>

kern_return_t port_allocate(task, port_name)
    task_t task;
    port_name_t *port_name; /* out */

Arguments
  task        The task in which the new port is created.
  port_name   The task's name for the new port.

Description
  port_allocate causes a port to be created for the specified task; the resulting port's name is returned in port_name. The target task initially has all three access rights to the port. If the caller is not the task specified by task, then it does not have any rights to the port. The new port is not a member of any port set.

Returns
  KERN_SUCCESS   The call succeeded.
  KERN_INVALID_ARGUMENT
                 task was invalid.
  KERN_RESOURCE_SHORTAGE
                 The kernel ran out of memory.

See Also
  port_deallocate
port_deallocate
#include <mach.h>

kern_return_t port_deallocate(task, port_name)
    task_t task;
    port_name_t port_name;

Arguments
    task    The task from which to remove the port rights.
    port_name    task’s name for the rights to be removed.

Description
    port_deallocate requests that the target task’s rights for a port be removed.

    If task has receive rights for the port, and the port is a member of a port set, the port is removed from
    the port set.

    If the target task is both the receiver and owner for the port, then the port is destroyed and all other
    tasks with send access are notified of the port’s destruction. If the task is only the receiver for the port,
    receive rights are sent to the owner. If the task is only the owner of the port, ownership rights are sent to
    the receiver.

Returns
    KERN_SUCCESS    The call succeeded.
    KERN_INVALID_ARGUMENT    task was invalid or port_name does not name a valid port.

See Also
    port_allocate
port_status
#include <mach.h>

kern_return_t port_status(task, port_name, enabled,
               num_msgs, backlog, owner, receiver)
    task_t task;
    port_name_t port_name;
    port_set_name_t *enabled; /* out */
    int *num_msgs; /* out */
    int *backlog; /* out */
    boolean_t *owner; /* out */
    boolean_t *receiver; /* out */

Arguments

    task The task owning the port right in question.
    port_name task’s name for the port right.
    enabled Returns task’s name for the port set which the named port belongs to, or
                  PORT_NULL if it isn’t in a set.
    num_msgs The number of messages queued on this port.
    backlog The number of messages which may be queued to this port without causing
            the sender to block.
    owner Returned as true iff the task is the owner of the port.
    receiver Returned as true iff the task is the receive of the port.

Description

    port_status returns the current status associated with task’s port right named port_name. If
receiver isn’t true, then the enabled, num_msg, and backlog arguments don’t return anything
meaningful.

Returns

    KERN_SUCCESS The call succeeded.
    KERN_INVALID_ARGUMENT task was invalid or port_name does not name a valid port.

See Also

    port_set_backlog, port_set_status
port_set_backlog
#include <mach.h>

kern_return_t port_set_backlog(task, port_name, backlog)
    task_t task;
    port_name_t port_name;
    int backlog;

Arguments
    task          The task owning the named port right.
    port_name     task's name for the port right.
    backlog       The new backlog to be set.

Description
    The port's backlog value is the number of unreceived messages that are allowed in its message queue before the kernel will refuse to accept any more sends to that port. port_set_backlog changes the backlog value on the specified port.

    task must have receive rights for the named port.

    The file <mach/mach_param.h> exports the system default value for a port's backlog as the constant PORT_BACKLOG_DEFAULT and the maximum backlog value as the constant PORT_BACKLOG_MAX.

Returns
    KERN_SUCCESS    The call succeeded.
    KERN_NOT_RECEIVER  port_name doesn't name receive rights in task.
    KERN_INVALID_ARGUMENT  task was invalid or port_name does not name a valid port or the desired backlog was non-positive or the desired backlog was greater than PORT_BACKLOG_MAX.

See Also
    msg_send, port_status
port_set_backup
#include <mach.h>

kern_return_t port_set_backup(task, primary, backup, previous)
    task_t task;
    port_name_t primary;
    port_t backup;
    port_t *previous;  /* out */

Arguments
    task         The task owning the named port right.
    primary      task’s name for the primary port.
    backup       The new backup port to be set.
    previous     The previous backup port.

Description
    A backup port provides an automatic mechanism to transfer port receive rights to another task or thread
in the event of a primary port’s attempted death. To be more precise, if a primary port has a backup port,
and the primary would have been destroyed by the deallocation of its receive rights, then instead the
receive right for the primary port is sent in a notify message (NOTIFY_PORT_DESTROYED) to the backup
port.

    A newly allocated port does not have a backup port. The port_set_backup call changes the backup
of the primary port. The target task must hold receive rights for the primary port. The caller supplies
send rights for the new backup port to which notification will be sent. The caller receives send rights for
the previous backup port or PORT_NULL if the target did not have a backup. port_set_backup
works atomically, so that if one backup port is exchanged for another, the primary port is never left without
a backup.

    When the primary port is sent in a notify message to the backup port, the primary port is left without a
backup port. When the task receives the notification and the receive rights to the primary port, it may
wish to use port_set_backup to reestablish the same or a different backup port. If the backup port is
destroyed before the primary, then the primary port is left without a backup. (A subsequent
port_set_backup call would return PORT_NULL).

Returns
    KERN_SUCCESS    The call succeeded.
    KERN_NOT_RECEIVER primary doesn’t name receive rights in task.
    KERN_INVALID_ARGUMENT task was invalid or primary or backup do not name a valid port.

See Also
    port_deallocate
port_set_allocate
#include <mach.h>

kern_return_t port_set_allocate(task, set_name)
    task_t task;
    port_set_name_t *set_name; /* out */

Arguments
    task               The task in which the new port set is created.
    set_name           The task's name for the new port set.

Description
    port_set_allocate causes a port set to be created for the specified task; the resulting set's name is returned in set_name. The new port set is empty.

Returns
    KERN_SUCCESS      The call succeeded.
    KERN_INVALID_ARGUMENT
                    task was invalid.
    KERN_RESOURCE_SHORTAGE
                    The kernel ran out of memory.

See Also
    port_set_deallocate
port_set_deallocate
#include <mach.h>

kern_return_t port_set_deallocate(task, set_name)
    task_t task;
    port_set_name_t set_name;

Arguments
    task       The task owning the port set to be destroyed.
    set_name   task’s name for the doomed port set.

Description
    port_set_deallocate requests that the target task’s port set be destroyed.

If the port set is non-empty, any members are first removed.

Returns
    KERN_SUCCESS   The call succeeded.
    KERN_FAILURE   set_name is task’s port set used for implementing the obsolete
                    port_enable and port_disable calls.
    KERN_INVALID_ARGUMENT task was invalid or set_name does not name a valid port set.

See Also
    port_set_allocate
port_set_add
#include <mach.h>

kern_return_t port_set_add(task, set_name, port_name)
    task_t task;
    port_set_name_t set_name;
    port_name_t port_name;

Arguments
    task                     The task owning the port set and port right.
    set_name                 task's name for the port set.
    port_name                task's name for the port.

Description
    port_set_add moves the named port into the named port set. task must have receive rights for the port.

    If the port is already a member of another port set, it is removed from that set first.

Returns
    KERN_SUCCESS             The call succeeded.
    KERN_NOT_RECEIVER        port_name doesn't name receive rights in task.
    KERN_INVALID_ARGUMENT    task was invalid or set_name does not name a valid port set or
                              port_name does not name a valid port.

See Also
    port_set_remove
port_set_remove
#include <mach.h>

kern_return_t port_set_remove(task, port_name)
    task_t task;
    port_name_t port_name;

Arguments
    task The task owning the receive rights and port set.
    port_name task’s name for the receive rights to be removed.

Description
    port_set_remove removes the named port from a port set. task must have receive rights for the
port, and the port must be a member of a port set.

Returns
    KERN_SUCCESS The call succeeded.
    KERN_NOT_RECEIVER
        port_name doesn’t name receive rights in task.
    KERN_NOT_IN_SET
        The port isn’t a member of a set.
    KERN_INVALID_ARGUMENT
        task was invalid or port_name does not name a valid port.

See Also
    port_set_add
port_set_status
#include <mach.h>

kern_return_t port_set_status(task, set_name, members, membersCnt)
    task_t task;
    port_set_name_t set_name;
    port_name_array_t *members; /* out array */
    unsigned int *membersCnt; /* out */

Arguments
    task       The task whose port set is queried.
    set_name   task’s name for the port set.
    members    task’s names for the port set’s members.
    membersCnt The number of port names returned.

Description
    port_set_status returns the members of a port set. members is an array that is automatically
    allocated when the reply message is received. The user may wish to vm_deallocate it when the data is
    no longer needed.

Returns
    KERN_SUCCESS  The call succeeded.
    KERN_INVALID_ARGUMENT
                  task was invalid or set_name does not name a valid port set.

See Also
    port_status
port_insert

#include <mach.h>

kern_return_t port_insert_send(task, my_port, his_name)
   task_t task;
   port_t my_port;
   port_name_t his_name;

kern_return_t port_insert_receive(task, my_port, his_name)
   task_t task;
   port_t my_port;
   port_name_t his_name;

Arguments

   task       The task getting the new rights.
   my_port    Rights supplied by the caller.
   his_name   The name by which task will know the new rights.

Description

port_insert_send and port_insert_receive give a task rights with a specific name. If task already has rights named his_name, or has some other name for my_port, then the operation will fail. his_name can't be a distinguished value like PORT_NULL.

port_insert_send inserts send rights, and port_insert_receive inserts receive and ownership rights.

Returns

   KERN_SUCCESS   The call succeeded.
   KERN_NAME_EXISTS   task already has a right named his_name.
   KERN_FAILURE   task already has rights to my_port.
   KERN_INVALID_ARGUMENT   task was invalid or new_name was an invalid name.

Notes

There is no way to insert just receive rights or just ownership rights.

See Also

port_extract_send, port_extract_receive
port_extract
#include <mach.h>

kern_return_t port_extract_send(task, his_name, his_port)
  task_t task;
  port_name_t his_name;
  port_t *his_port; /* out */

kern_return_t port_extract_receive(task, his_name, his_port)
  task_t task;
  port_name_t his_name;
  port_t *his_port; /* out */

Arguments
  task       The task whose rights the caller takes.
  his_name   The name by which task knows the rights.
  his_port   Rights returned to the caller.

Description
  port_extract_send and port_extract_receive remove task’s rights for a port and return the
  rights to the caller. task is left with no rights for the port.

  port_extract_send extracts send rights; task can’t have receive or ownership rights for the named
  port. port_extract_receive extracts receive/ownership rights, both of which task must hold.

Returns
  KERN_SUCCESS   The call succeeded.
  KERN_INVALID_ARGUMENT
                   task was invalid or his_name does not name a port for which task has the
                   required rights.

Notes
  There is no way to extract just receive rights or just ownership rights.

See Also
  port_insert_send, port_insert_receive
4. Task and thread primitives

4.1. Basic terms

The MACH system separates the traditional notion of a process into two subconcepts:

- **Tasks** contain the capabilities, namely the port rights, resource limits, and address space of a running entity. Tasks perform no computation; they are a framework for running threads.
- **Threads** contain the minimal processing state associated with a computation, e.g. a program counter, a stack pointer, and a set of registers. A thread exists within exactly one task; however, one task may contain many threads.

Tasks are the basic unit of protection. All threads within a task have access to all of that task’s capabilities, and are thus not protected from each other.

Threads are the basic unit of scheduling. On a multiprocessor host, multiple threads from one task may be executing simultaneously (within the task’s one address space). A thread may be in a suspended state (prevented from running), or in a runnable state (may be running or be scheduled to run). There is a non-negative suspend count associated with each thread. The suspend count is zero for runnable threads and positive for suspended threads.

Tasks may be suspended or resumed as a whole. A thread may only execute when both it and its task are runnable. Resuming a task does not cause all component threads to begin executing, but only those threads which are not suspended.

Both tasks and threads are represented by ports. These ports are called the **task kernel port** and the **thread kernel port**. These are the handles that are used in the task and thread kernel calls to identify to the kernel which task or thread is to be affected by the call. The two primitives task_self() and thread_self() return the task and thread ports of the currently executing thread. Tasks may have access to the task and thread ports of other tasks and threads. For example, a task that creates another task or thread gets access to the new task or thread port. Also any thread may pass access to these ports in a message to another thread in the same or different task. Having access to a task or thread port enables the possessor to perform kernel calls on behalf of that task or thread. Access to a task’s kernel port indirectly permits access to all threads within that task via the task_threads call; however, access to a thread’s kernel port does not currently imply access to its task port.

In addition to their kernel ports, tasks and threads have a number of special ports associated with them. In general these are ports that the kernel must know about in order to communicate with the task or thread in a structured manner.

There are three ports associated with a task in addition to its kernel port:

- The **notify port**, on which the task should attempt to receive notification of such kernel events as the destruction of a port to which it has send rights. The task has receive rights to this port and can get its value from the primitive task_notify().
- The **exception port**, to which the kernel sends messages when an exception occurs. Exceptions are synchronous interruptions to the normal flow of program control caused by the program itself. They include illegal memory accesses, protection violations, arithmetic exceptions, and hardware instructions intended to support emulation, debugging and/or error detection. Some of these exceptions are handled transparently by the operating system but
some must be reported to the user program. A default exception port is inherited from the parent at task creation time. This port can be changed by the task or any one of its threads in order to take an active role in handling exceptions.

- The **bootstrap port**, to which a new task can send a message that will return any other system service ports that the task needs, for example a port to the Network Nameserver or the Environment Manager. Send rights to this port are inherited from the parent at task creation. This is the one port that the kernel does not actually use, it just makes it available to a new task.

There are two ports associated with a thread in addition to its kernel port:

- The **thread reply port**, which may be used for initial messages from a parent or for early remote procedure calls. The `thread_reply()` primitive returns receive rights to this port.

- The **thread exception port**, to which kernel sends exceptions occurring in this thread. This port is set to `PORT_NULL` at thread creation and can be set subsequently by the call `thread_set_exception_port`. As long as the thread exception port is `PORT_NULL` the task exception port will be used instead.

### 4.2. Access to Tasks: Terminology

In this and following sections, calls are described which may manipulate the state of a task. Although some of the descriptions may refer to tasks as performing these calls, it is in fact some thread within a task which makes any call.

Furthermore, any thread within any task which holds access rights to that task (i.e. task kernel port) may perform calls which take a task as an argument. Customarily, only threads within a task will manipulate that task’s state, but this custom is not enforced by the MACH kernel. Debugger tasks are a notable exception to this rule. Similarly, access to a thread is controlled by access to its thread kernel port.
task_create

#include <mach.h>

kern_return_t task_create(parent_task, inherit_memory, child_task)
    task_t parent_task
    boolean_t inherit_memory;
    task_t *child_task; /* out */

Arguments

target_task The task from which the child's capabilities are drawn.

inherit_memory If set, the child task's address space is built from the parent task according to its memory inheritance values; otherwise, the child task is given an empty address space.

child_task The new task.

Description

 task_create creates a new task from parent_task; the resulting task (child_task) acquires shared or copied parts of the parent's address space (see vm_inherit). The child task initially contains no threads.

    The child task gets the four special ports created or copied for it at task creation. The task_kernel_port is created and send rights for it are given to the child and returned to the caller. The task_notify_port is created and receive, ownership and send rights for it are given to the child. The caller has no access to it. The task_bootstrap_port and the task_exception_port are inherited from the parent task. The new task can get send rights to these ports with the call task_get_special_port.

Returns

 KERN_SUCCESS A new task has been created.
 KERN_INVALID_ARGUMENT parent_task is not a valid task port.
 KERN_RESOURCE_SHORTAGE Some critical kernel resource is unavailable.

See Also

 task_terminate, task_suspend, task_resume, task_special_ports, task_threads, thread_create, thread_resume, vm_inherit

Notes

 Not implemented yet. Use fork.
task_terminate
#include <mach.h>

kern_return_t task_terminate(target_task)
    task_t target_task;

Arguments
    target_task   The task to be destroyed.

Description
    task_terminate destroys the task specified by target_task and all its threads. All resources that are used only by this task are freed. Any port to which this task has receive and ownership rights is destroyed.

Returns
    KERN_SUCCESS   The task has been killed.
    KERN_INVALID_ARGUMENT
target_task is not a task.

See Also
    task_create, task_suspend, task_resume, thread_terminate, thread_suspend

Notes
    Not implemented yet.
task_suspend
#include <mach.h>

kern_return_t task_suspend(target_task)
    task_t target_task;

Arguments
    target_task    The task to be suspended.

Description
    Increments the task's suspend count and stops all threads in the task. As long as the suspend count is
    positive newly created threads will not run. This call does not return until all threads are suspended.

    The count may become greater than one, with the effect that it will take more than one resume call to
    restart the task.

Returns
    KERN_SUCCESS    The task has been suspended.
    KERN_INVALID_ARGUMENT
        target_task is not a task.

See Also
    task_create, task_terminate, task_resume, task_info, thread_suspend
task_resume
#include <mach.h>

kern_return_t task_resume(target_task)
    task_t target_task;

Description
Decrements the task's suspend count. If it becomes zero, all threads with zero suspend counts in the
task are resumed. The count may not become negative.

Arguments
    target_task  The task to be resumed.

Returns
    KERN_SUCCESS  The task has been resumed.
    KERN_FAILURE  The suspend count is already at zero.
    KERN_INVALID_ARGUMENT  target_task is not a task.

See Also
    task_create, task_terminate, task_suspend, task_info, thread_suspend,
    thread_resume, thread_info
task_special_ports
#include <mach.h>

kern_return_t task_get_special_port(task, which_port, special_port)
    task_t task;
    int which_port;
    port_t *special_port; /* out */

kern_return_t task_set_special_port(task, which_port, special_port)
    task_t task;
    int which_port;
    port_t special_port;

task_t task_self()

port_t task_notify()

Arguments
  task The task for which to get the port
  which_port the port that is requested. Is one of TASK_NOTIFY_PORT,
              TASK_BOOTSTRAP_PORT, TASK_EXCEPTION_PORT.
  special_port the value of the port that is being requested or being set.

Description
  get_special_port returns send rights to one of a set of special ports for the task specified by task.
  In the case of the task’s own task_notify_port, the task also gets receive and ownership rights.

  set_special_port sets one of a set of special ports for the task specified by task.

  task_self returns the port to which kernel calls for the currently executing thread should be directed.
  Currently, task_self returns the task kernel port which is a port for which the kernel has receive rights
  and which it uses to identify a task. In the future it may be possible for one task to interpose a port as
  another’s task’s kernel port. At that time, task_self will still return the port to which the executing thread
  should direct kernel calls, but it may no longer be a port on which the kernel has receive rights.

  If one task, the controller, has send access to the kernel port of another task, the subject task, then the
  controller task can perform kernel operations for the subject task. Normally only the task itself and the
  task that created it will have access to the task kernel port, but any task may pass rights to its kernel port
  to any other task.

  task_notify returns receive, ownership and send rights to the notify port associated with the task
  to which the executing thread belongs. The notify port is a port on which the task should receive
  notification of such kernel events of the destruction of a port to which it has send rights.

  The other special ports associated with a task are the bootstrap port and the exception port. The
  bootstrap port is a port to which a thread may send a message requesting other system service ports.
  This port is not used by the kernel. The task’s exception port is the port to which messages are sent by
the kernel when an exception occurs and the thread causing the exception has no exception port of its own.

Within the C environment, task_self and task_notify are implemented as macros which execute the system traps the first time and thereafter return a cached value for the ports. Thus it is unnecessary for a programmer to cache these variables himself and such caching may interfere with the future implementation of port interposition.

The following macros to call task_set/get_special_port for a specific port are defined in <mach/task_special_ports.h>: task_get_notify_port, task_set_notify_port, task_get_exception_port, task_set_exception_port, task_get_bootstrap_port and task_set_bootstrap_port.

Returns

KERN_SUCCESS The port was returned or set.
KERN_INVALID_ARGUMENT task is not a task or which_port is an invalid port selector.

See Also

thread_special_ports,mach_init,task_create

Notes

The call on the bootstrap port to get system service ports has not been implemented yet.

TASK_KERNEL_PORT may be added to the set of ports that task_set_special_port accepts.
task_info

#include <mach.h>

/* the definition of task_info_t from mach.h - mach/task_info.h is */
typedef int *task_info_t; /* variable length array of int */

/* currently the only interpretation of info is */

struct task_basic_info {
    int suspend_count; /* suspend count for task */
    int base_priority; /* base scheduling priority */
    vm_size_t virtual_size; /* number of virtual pages */
    vm_size_t resident_size; /* number of resident pages */
    time_value_t user_time; /* total user run time for
        terminated threads */
    time_value_t system_time; /* total system run time for
        terminated threads */
};
typedef struct task_basic_info *task_basic_info_t;

kern_return_t task_info(target_task, flavor, task_info, task_infoCnt)
    task_t target_task;
    int flavor;
    task_info_t task_info; /* in and out */
    unsigned int *task_infoCnt; /* in and out */
Arguments

target_task The task to be affected.
flavor The type of statistics that are wanted. Currently only TASK_BASIC_INFO is
    implemented.
task_info Statistics about the task specified by target_task.
task_infoCnt Size of the info structure. Currently only TASK_BASIC_INFO_COUNT is
    implemented.

Description

Returns the selected information array for a task, as specified by flavor. task_info is an array of
    integers that is supplied by the caller, and filled with specified information. task_infoCnt is supplied as
    the maximum number of integers in task_info. On return, it contains the actual number of integers in
    task_info.

Currently there is only one flavor of information which is defined by TASK_BASIC_INFO. Its size is
    defined by TASK_BASIC_INFO_COUNT.

Returns

KERN_SUCCESS The call succeeded.
KERN_INVALID_ARGUMENT
    target_task is not a task or flavor is not recognized.
MIG_ARRAY_TOO_LARGE
    Returned info array is too large for task_info. task_info is filled as much
    as possible. task_infoCnt is set to the number of elements that would be
    returned if there were enough room.
See Also

  task_special_ports, task_threads, thread_info, thread_state
task_threads
#include <mach.h>

kern_return_t task_threads(target_task, thread_list, thread_count)
    task_t target_task;
    thread_array_t *thread_list; /* out, ptr to array */
    int *thread_count; /* out */

Arguments
    target_task The task to be affected.
    thread_list The set of threads contained within target_task; no particular ordering is
                guaranteed.
    thread_count The number of threads in the thread_list.

Description
    task_threads gets send rights to the kernel port for each thread contained in target_task.
    thread_list is an array that is created as a result of this call. The caller may wish to vm_deallocate
    this array when the data is no longer needed.

Returns
    KERN_SUCCESS The call succeeded.
    KERN_INVALID_ARGUMENT target_task is not a task.

See Also
    thread_create, thread_terminate, thread_suspend
thread_create
#include <mach.h>

kern_return_t thread_create(parent_task, child_thread)
    task_t parent_task;
    thread_t *child_thread; /* out */

Description

thread_create creates a new thread within the task specified by parent_task. The new thread has no processor state, and has a suspend count of 1. To get a new thread to run, first thread_create is called to get the new thread’s identifier, (child_thread). Then thread_set_state is called to set a processor state, and finally thread_resume is called to get the thread scheduled to execute.

When the thread is created send rights to its thread kernel port are given to it and returned to the caller in child_thread. The new thread’s exception port is set to PORT_NULL.

Arguments

parent_task The task which is to contain the new thread.
child_thread The new thread.

Returns

KERN_SUCCESS A new thread has been created.
KERN_INVALID_ARGUMENT parent_task is not a valid task.
KERN_RESOURCE_SHORTAGE Some critical kernel resource is not available.

See Also

task_create, task_threads, thread_terminate, thread_suspend, thread_resume, thread_special_ports, thread_set_state
thread_terminate
#include <mach.h>

kern_return_t thread_terminate(target_thread)
    thread_t target_thread;

Arguments
    target_thread  The thread to be destroyed.

Description
    thread_terminate destroys the thread specified by target_thread.

Returns
    KERN_SUCCESS     The thread has been killed.
    KERN_INVALID_ARGUMENT
        target_thread is not a thread.

See Also
    task_terminate, task_threads, thread_create, thread_resume, thread_suspend
thread_suspend
#include <mach.h>

kern_return_t thread_suspend(target_thread);
    thread_t target_thread;

Arguments
    target_thread  The thread to be suspended.

Description
Increments the thread's suspend count and prevents the thread from executing any more user level
instructions. In this context a user level instruction is either a machine instruction executed in user mode
or a system trap instruction including page faults. Thus if a thread is currently executing within a system
trap the kernel code may continue to execute until it reaches the system return code or it may suspend
within the kernel code. In either case, when the thread is resumed the system trap will return. This could
cause unpredictable results if the user did a suspend and then altered the user state of the thread in order
to change its direction upon a resume. The call thread_abort is provided to allow the user to abort any
system call that is in progress in a predictable way.

The suspend count may become greater than one with the effect that it will take more than one resume
call to restart the thread.

Returns
    KERN_SUCCESS     The thread has been suspended.
    KERN_INVALID_ARGUMENT
        target_thread is not a thread.

See Also
    task_suspend, task_resume, thread_info, thread_state, thread_resume,
    thread_terminate, thread_abort
thread_resume
#include <mach.h>

kern_return_t thread_resume(target_thread)
    thread_t target_thread;

Arguments
    target_thread  The thread to be resumed.

Description
    Decrements the threads's suspend count. If the count becomes zero the thread is resumed. If it is still 
    positive, the thread is left suspended. The suspend count may not become negative.

Returns
    KERN_SUCCESS    The thread has been resumed.
    KERN_FAILURE    The suspend count is already zero.
    KERN_INVALID_ARGUMENT    target_thread is not a thread.

See Also
    task_suspend, task_resume thread_info, thread_create, thread_terminate, thread_suspend
thread_abort

#include <mach.h>

kern_return_t thread_abort(target_thread)
    thread_t target_thread;

Arguments

target_thread  The thread to be interrupted.

Description

thread_abort aborts the kernel primitives: msg_send, msg_receive and msg_rpc and page-faults, making the call return a code indicating that it was interrupted. The call is interrupted whether or not the thread (or task containing it) is currently suspended. If it is suspended, the thread receives the interrupt when it is resumed. This call also aborts any priority depression caused by the DEPRESS option to thread_switch.

A thread will retry an aborted page-fault if its state is not modified before it is resumed. msg_send returns SEND_INTERRUPTED; msg_receive returns RCV_INTERRUPTED; msg_rpc returns either SEND_INTERRUPTED or RCV_INTERRUPTED, depending on which half of the RPC was interrupted.

The main reason for this primitive is to allow one thread to cleanly stop another thread in a manner that will allow the future execution of the target thread to be controlled in a predictable way. thread_suspend keeps the target thread from executing any further instructions at the user level, including the return from a system call. thread_get/set_state allows the examination or modification of the user state of a target thread. However, if a suspended thread was executing within a system call, it also has associated with it a kernel state. This kernel state can not be modified by thread_set_state with the result that when the thread is resumed the system call may return changing the user state and possibly user memory. thread_abort aborts the kernel call from the target thread’s point of view by resetting the kernel state so that the thread will resume execution at the system call return with the return code value set to one of the interrupted codes. The system call itself will either be entirely completed or entirely aborted, depending on the precise moment at which the abort was received. Thus if the thread’s user state has been changed by thread_set_state, it will not be modified by any unexpected system call side effects.

For example to simulate a Unix signal, the following sequence of calls may be used:

    thread_suspend Stops the thread

    thread_abort Interrupts any system call in progress, setting the return value to ‘interrupted’. Since the thread is stopped, it will not return to user code.

    thread_set_state Alters thread’s state to simulate a procedure call to the signal handler

    thread_resume Resumes execution at the signal handler. If the thread’s stack has been correctly set up, the thread may return to the interrupted system call.

(of course, the code to push an extra stack frame and change the registers is VERY machine-
Calling thread_abort on a non-suspended thread is pretty risky, since it is very difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

**Returns**

- KERN_SUCCESS: The thread received an interrupt
- KERN_INVALID_ARGUMENT: target_thread is not a thread.

**See Also**

thread_info, thread_state, thread_terminate, thread_suspend, thread_switch
thread_special_ports
#include <mach.h>

kern_return_t thread_get_special_port(thread, which_port, special_port)
    thread_t thread;
    int which_port;
    port_t *special_port;

kern_return_t thread_set_special_port(thread, which_port, special_port)
    thread_t thread;
    int which_port;
    port_t special_port;

thread_t thread_self()

port_t thread_reply()

Arguments
    thread The thread for which to get the port
    which_port the port that is requested. Is one of THREAD_REPLY_PORT or
                 THREAD_EXCEPTION_PORT.
    special_port the value of the port that is being requested or being set.

Description
    get_special_port returns send rights to one of a set of special ports for the thread specified by
    thread. In the case of getting the thread’s own thread_reply_port, receive and ownership rights are
    also given to the thread.

    set_special_port sets one of a set of special ports for the thread specified by thread.

    thread_self returns the port to which kernel calls for the currently executing thread should be
directed. Currently, thread_self returns the thread kernel port which is a port for which the kernel
has receive rights and which it uses to identify a thread. In the future it may be possible for one thread to
interpose a port as another’s thread’s kernel port. At that time, thread_self will still return the port to
which the executing thread should direct kernel calls, but it may no longer be a port on which the kernel
has receive rights.

    If one thread, the controller, has send access to the kernel port of another thread, the subject thread,
then the controller thread can perform kernel operations for the subject thread. Normally only the thread
itself and its parent task will have access to the thread kernel port, but any thread may pass rights to its
kernel port to any other thread.

    thread_reply returns receive, ownership and send rights to the reply port of the calling thread. The
reply port is a port to which the thread has receive rights. It is used to receive any initialization messages
and as a reply port for early remote procedure calls.

    The following macros to call thread_get/set_special_port for a specific port are defined in
<mach/thread_special_ports.h>: thread_get_reply_port, thread_set_reply_port,
thread_get_exception_port and thread_set_exception_port.
A thread also has access to its task’s special ports.

**Returns**

- **KERN_SUCCESS**  The port was returned or set.
- **KERN_INVALID_ARGUMENT**  thread is not a thread or which_port is an invalid port selector.

**See Also**

task_special_ports, thread_create

**Notes**

THREAD_KERNEL_PORT may be added to the set of ports that thread_set_special_port accepts.
thread_info
#include <mach.h>

/* the definition of thread_info_data_t from mach.h - mach/thread_info.h is */
typedef int *thread_info_t; /* variable length array of int */

/* only current interpretation of thread_info */

struct thread_basic_info {
    time_value_t user_time; /* user run time */
    time_value_t system_time; /* system run time */
    int cpu_usage; /* scaled cpu usage percentage */
    int base_priority; /* base scheduling priority */
    int cur_priority; /* current scheduling priority */
    int run_state; /* run state (see below) */
    int flags; /* various flags (see below) */
    int suspend_count; /* suspend count for thread */
    long sleep_time; /* number of seconds that thread has been sleeping */
};
typedef struct thread_basic_info *thread_basic_info_t;

The possible values of the run_state field are:
TH_STATE_RUNNING, thread is running normally
TH_STATE_STOPPED, thread is suspended
TH_STATE_WAITING, thread is waiting normally
TH_STATE_UNINTERRUPTIBLE, thread is in an uninterruptible wait
TH_STATE_HALTED, thread is halted at a clean point

The possible values of the flags field are:
TH_FLAGS_SWAPPED, thread is swapped out
TH_FLAGS_IDLE, thread is an idle thread

kern_return_t thread_info(target_thread, flavor, thread_info,
                          thread_infoCnt)
thread_t       target_thread;
int            flavor;
thread_info_t  thread_info;   /* in and out */
unsigned int   *thread_infoCnt;   /* in and out */

Arguments

    target_thread  The thread to be affected.
    flavor         The type of statistics that are wanted. Currently only THREAD_BASIC_INFO
                    is implemented.
    thread_info    Statistics about the thread specified by target_thread.
    thread_infoCnt Size of the info structure. Currently only THREAD_BASIC_INFO_COUNT is
                    implemented.

Description

Returns the selected information array for a thread, as specified by flavor. thread_info is an array
of integers that is supplied by the caller and returned filled with specified information. thread_infoCnt
is supplied as the maximum number of integers in thread_info. On return, it contains the actual
number of integers in thread_info.
Currently there is only one flavor of information which is defined by THREAD_BASIC_INFO. Its size is defined by THREAD_BASIC_INFO_COUNT.

**Returns**

- **KERN_SUCCESS**  The call succeeded.
- **KERN_INVALID_ARGUMENT**  target_thread is not a thread or flavor is not recognized.
- **MIG_ARRAY_TOO_LARGE**  Returned info array is too large for thread_info. thread_info is filled as much as possible. thread_infoCnt is set to the number of elements that would have been returned if there were enough room.

**See Also**

- thread_special_ports, task_threads, task_info, thread_state
thread_state
#include <mach.h>

kern_return_t thread_get_state(target_thread, flavor, old_state,
   old_stateCnt)
   thread_t target_thread;
   int flavor;
   thread_state_data_t old_state; /* in and out */
   unsigned int *old_stateCnt; /* in and out */

kern_return_t thread_set_state(target_thread, flavor, new_state,
   new_stateCnt)
   thread_t target_thread;
   int flavor;
   thread_state_data_t new_state;
   unsigned int new_stateCnt;

Arguments
   target_thread thread to get or set the state for.
   flavor The type of state that is to be manipulated. Currently must be one of the
   following values: VAX_THREAD_STATE, ROMP_THREAD_STATE,
   SUN_THREAD_STATE_REGS, SUN_THREAD_STATE_FPA
   new_state an array of state information
   old_state an array of state information
   new_stateCnt the size of the state information array. Currently must be one of the following
   values: VAX_THREAD_STATE_COUNT, ROMP_THREAD_STATE_COUNT,
   SUN_THREAD_STATE_REGS_COUNT, SUN_THREAD_STATE_FPA_COUNT
   old_stateCnt same as new_stateCnt

Description
   thread_get_state returns the state component (e.g. the machine registers) of target_thread as
   specified by flavor. The old_state is an array of integers that is provided by the caller and returned
   filled with the specified information. old_stateCnt is input set to the maximum number of integers in
   old_state and returned equal to the actual number of integers in old_state.

   thread_set_state sets the state component (e.g. the machine registers) of target_thread as
   specified by flavor. The new_state is an array of integers. new_stateCnt is the number of
   elements in new_state. The entire set of registers is reset. This will do unpredictable things if
   target_thread is not suspended.

   target_thread may not be thread_self for either of these calls.

The definition of the state structures can be found in <machine/thread_status.h>

Returns
   KERN_SUCCESS The state has been set or returned
   MIG_ARRAY_TOO_LARGE Returned state is too large for the new_state array. new_state is filled in
   as much as possible and new_stateCnt is set to the number of elements
   that would be returned if there were enough room.
KERN_INVALID_ARGUMENT
target_thread is not a thread or is thread_self or flavor is
unrecognized for this machine.

See Also
task_info, thread_info
5. Virtual memory primitives

5.1. Basic terms

Each MACH task has a large virtual address space within which its threads execute. A virtual address space is divided into fixed size pages. The size of a virtual page is set at system initialization and may differ on different machines. A virtual address space may be sparse, that is, there may be ranges of addresses which are not allocated followed by ranges that are allocated.

A task may allocate virtual memory in its address space; physical memory will be acquired only when necessary, and seldom-used memory may be paged to backing storage.

A region of an address space is that memory associated with a continuous range of addresses; that is, a start address and an end address. The MACH kernel will extend regions to include entire virtual memory pages containing the first and last address in a specified range. Regions consist of pages which have different protection or inheritance characteristics.

A task may protect the virtual pages of its address space to allow/prevent access to that memory. The current protection is used to determine the access rights of an executing thread. In addition, a maximum protection value limits the current protection.

A task may specify that pages of its address space be inherited by child tasks in one of three ways: shared, copied, or absent. Inheritance may be changed at any time; only at the time of task creation is inheritance information used. The only way two MACH tasks can share the same physical memory is for one of the tasks to inherit shared access to memory from a parent. When a child task inherits memory from a parent, it gets the same protection on that memory that its parent had.

Protection and inheritance is attached to a task’s address space, not the physical memory contained in that address space. Tasks which share memory may specify different protection or inheritance for their shared regions.

Physical pages in an address space have paging objects associated with them. These objects identify the backing storage to be used when a page is to be read in as the result of a reference or written to in order to free physical memory. A paging object is identified outside of the kernel by an unforgeable identifier (implemented as a port which is only used for identification and not message transmission), and inside the kernel by a data transmission port, that will respond to get and put page calls.

In addition to memory explicitly allocated using vm_allocate, memory may appear in a task’s address space as the result of a msg_receive operation.
vm_allocate
#include <mach.h>

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

Arguments

    target_task  Task whose virtual address space is to be affected.
    address      Starting address. If the anywhere option is false, an attempt is made to
                 allocate virtual memory starting at this virtual address. If this address
                 is not at the beginning of a virtual page, it will be rounded down to one. If
                 there is not enough space at this address, no memory will be allocated. If
                 the anywhere option is true, the input value of this address will be ignored,
                 and the space will be allocated wherever it is available. In either case, the
                 address at which memory was actually allocated will be returned in address.
    size         Number of bytes to allocate (rounded by the system in a machine dependent
                 way to an integral number of virtual pages).
    anywhere     If true, the kernel should find and allocate any region of the specified size,
                 and return the address of the resulting region in address. If false, virtual
                 memory will be allocated starting at address, rounded to a virtual page
                 boundary if there is sufficient space.

Description

    vm_allocate allocates a region of virtual memory, placing it in the specified task’s address space. The
    physical memory is not actually allocated until the new virtual memory is referenced. By default, the
    kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest
    page size. The global variable vm_page_size contains the page size. task_self_ returns the value
    of the current task port which should be used as the target_task argument in order to allocate memory
    in the caller’s address space. For languages other than C, these values can be obtained by the calls
    vm_statistics and task_self. Initially, the pages of allocated memory will be protected to allow all
    forms of access, and will be inherited in child tasks as a copy. Subsequent calls to vm_protection and
    vm_inheritance may be used to change these properties. The allocated region is always zero-filled.

Returns

    KERN_SUCCESS  Memory allocated.
    KERN_INVALID_ADDRESS  Illegal address specified.
    KERN_NO_SPACE  Not enough space left to satisfy this request

See Also

    vm_deallocate, vm_inherit, vm_protect, vm_regions, vm_statistics, task_self

MACH 23 August 1990 MACH
vm_deallocate
#include <mach.h>

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

Arguments
    target_task Task whose virtual memory is to be affected.
    address Starting address (will be rounded down to a page boundary).
    size Number of bytes to deallocate (will be rounded up to give a page boundary).

Description
vm_deallocate relinquishes access to a region of a task’s address space, causing further access to that memory to fail. This address range will be available for reallocation. Note, that because of the rounding to virtual page boundaries, more than size bytes may be deallocated. Use vm_page_size or vm_statistics to find out the current virtual page size.

This call may be used to deallocate memory that was passed to a task in a message (via out of line data). In that case, the rounding should cause no trouble, since the region of memory was allocated as a set of pages.

The vm_deallocate call affects only the task specified by the target_task. Other tasks which may have access to this memory may continue to reference it.

Returns
    KERN_SUCCESS Memory deallocated.
    KERN_INVALID_ADDRESS Illegal or non-allocated address specified.

See Also
    vm_allocate, vm_statistics, msg_receive
vm_read
#include <mach.h>

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

Arguments

    target_task Task whose memory is to be read.
    address The first address to be read (must be on a page boundary).
    size The number of bytes of data to be read (must be an integral number of pages)
    data The array of data copied from the given task.
    data_count The size of the data array in bytes. (will be an integral number of pages).

Description

vm_read allows one task's virtual memory to be read by another task. Note that the data array is returned in a newly allocated region; the task reading the data should vm_deallocate this region when it is done with the data.

Returns

    KERN_SUCCESS Memory read.
    KERN_INVALID_ARGUMENT Either the address does not start on a page boundary or the size is not an integral number of pages.
    KERN_NO_SPACE There is not enough room in the callers virtual memory to allocate space for the data to be returned.
    KERN_PROTECTION_FAILURE The address region in the target task is protected against reading.
    KERN_INVALID_ADDRESS Illegal or non_allocated address specified, or there was not size bytes of data following that address.

See Also

vm_read, vm_write, vm_copy, vm_deallocate
# vm_write

#include <mach.h>

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

Arguments

target_task    Task whose memory is to be written.
address        Starting address in task to be affected (must be a page boundary).
data           An array of bytes to be written.
data_count     The size of the data array (must be an integral number of pages).

Description

vm_write allows a task's virtual memory to be written by another task. Use vm_page_size or
vm_statistics to find out the virtual page size.

Returns

KERN_SUCCESS   Memory written.
KERN_INVALID_ARGUMENT
    Either the address does not start on a page boundary or the size is not an
    integral number of pages.
KERN_PROTECTION_FAILURE
    The address region in the target task is protected against writing.
KERN_INVALID_ADDRESS
    Illegal or non_allocated address specified or there is not data_count of
    allocated memory starting at address.

See Also

vm_copy, vm_protect, vm_read, vm_statistics
vm_copy

#include <mach.h>

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

Arguments

    target_task   Task whose virtual memory is to be affected.
    source_address Address in target_task of the start of the source range (must be a page boundary).
    count         Number of bytes to copy (must be an integral number of pages).
    dest_address  Address in target_task of the start of the destination range (must be a page boundary).

Description

    vm_copy causes the source memory range to be copied to the destination address; the destination region may not overlap the source region. The destination address range must already be allocated and writable; the source range must be readable.

Returns

    KERN_SUCCESS  Memory copied.
    KERN_INVALID_ARGUMENT Either the address does not start on a page boundary or the size is not an integral number of pages.
    KERN_PROTECTION_FAILURE Either the destination region was not writable, or the source region was not readable.
    KERN_INVALID_ADDRESS  Illegal or non-allocated address specified or insufficient memory allocated at one of the addresses.

See Also

    vm_protect, vm_write, vm_statistics
vm_region

#include <mach.h>

kern_return_t vm_region(target_task, address, size, protection,
max_protection, inheritance, shared,
object_name, offset)

vm_task_t target_task;
vm_address_t *address; /* in/out */
vm_size_t *size; /* out */
vm_prot_t *protection; /* out */
vm_prot_t *max_protection; /* out */
vm_inherit_t *inheritance; /* out */
boolean_t *shared; /* out */
port_t *object_name; /* out */
vm_offset_t *offset; /* out */

Arguments

target_task The task for which an address space description is requested.
address The address at which to start looking for a region.
size The size (in bytes) of the located region.
protection The current protection of the region.
max_protection The maximum allowable protection for this region.
inheritance The inheritance attribute for this region.
shared Is this region shared or not.
object_name The port identifying the memory object associated with this region. (See pager_init.)
offset The offset into the pager object that this region begins at.

Description

vm_region returns a description of the specified region of the target task's virtual address space. vm_region begins at address and looks forward thru memory until it comes to an allocated region. (If address is within a region, then that region is used.) Various bits of information about the region are returned. If address was not within a region, then address is set to the start of the first region which follows the incoming value. In this way an entire address space can be scanned.

Returns

KERN_SUCCESS Region located and information returned.
KERN_NO_SPACE There is no region at or above address in the specified task.

See Also

vm_allocate, vm_deallocate, vm_protect, vm_inherit
vm_protect
#include <mach.h>

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

Arguments
  target_task  Task whose virtual memory is to be affected.
  address      Starting address (will be rounded down to a page boundary).
  size         Size in bytes of the region for which protection is to change (will be rounded
               up to give a page boundary).
  set_maximum  If set, make the protection change apply to the maximum protection
               associated with this address range; otherwise, the current protection on this
               range is changed.  If the maximum protection is reduced below the current
               protection, both will be changed to reflect the new maximum.
  new_protection A new protection value for this region; a set of: VM_PROT_READ,
                  VM_PROT_WRITE, VM_PROT_EXECUTE.

Description
  vm_protect sets the virtual memory access privileges for a range of allocated addresses in a task's
  virtual address space.  The protection argument describes a combination of read, write, and execute
  accesses that should be permitted.

  The enforcement of virtual memory protection is machine-dependent.  Some combinations of access
  rights may not be supported.  In particular, the kernel interface allows any of the following:  write
  permission may imply read permission; read permission may imply execute permission; or, execute
  permission may imply read permission.

  All architectures must support the following access combinations:  all (read, write, and execute) access;
  write-protected (read and execute) access; no access.

  For the Vax, RT/PC, and Sun3, all three of the reductions stated above apply.  That is: VM_PROT_WRITE
  allows read, execute and write access, VM_PROT_READ or VM_PROT_EXECUTE allows
  read and execute access, but not write access.

Returns
  KERN_SUCCESS   Memory protected.
  KERN_PROTECTION_FAILURE
     An attempt was made to increase the current or maximum protection beyond
     the existing maximum protection value.
  KERN_INVALID_ADDRESS
     Illegal or non-allocated address specified.
vm_inherit

#include <mach.h>

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

Arguments

target_task  Task whose virtual memory is to be affected.
address  Starting address (will be rounded down to a page boundary).
size  Size in bytes of the region for which inheritance is to change (will be rounded up to give a page boundary).
new_inheritance  How this memory is to be inherited in child tasks. Inheritance is specified by using one of these following three values:

VM_INHERIT_SHARE  Child tasks will share this memory with this task.
VM_INHERIT_COPY  Child tasks will receive a copy of this region.
VM_INHERIT_NONE  This region will be absent from child tasks.

Description

vm_inherit specifies how a region of a task’s address space is to be passed to child tasks at the time of task creation. Inheritance is an attribute of virtual pages, thus the addresses and size of memory to be set will be rounded out to refer to whole pages.

Setting vm_inherit to VM_INHERIT_SHARE and forking a child task is the only way two Mach tasks can share physical memory. Remember that all the theads of a given task share all the same memory.

Returns

KERN_SUCCESS  Memory protected.
KERN_INVALID_ADDRESS  Illegal address specified.

See Also

task_create, vm_regions
### vm_statistics

```c
#include <mach.h>

struct vm_statistics {
    long pagesize;   /* page size in bytes */
    long free_count; /* # of pages free */
    long active_count; /* # of pages active */
    long inactive_count; /* # of pages inactive */
    long wire_count; /* # of pages wired down */
    long zero_fill_count; /* # of zero fill pages */
    long reactivations; /* # of pages reactivated */
    long pageins; /* # of pageins */
    long pageouts; /* # of pageouts */
    long faults; /* # of faults */
    long cow_faults; /* # of copy-on-writes */
    long lookups; /* object cache lookups */
    long hits; /* object cache hits */
};

typedef struct vm_statistics vm_statistics_data_t;

kern_return_t vm_statistics(target_task, vm_stats)
    task_t target_task;
    vm_statistics_data_t *vm_stats; /* out */
```

#### Arguments

- **target_task**
  - Task which is requesting statistics.

- **vm_stats**
  - The structure that will receive the statistics.

#### Description

The `vm_statistics` function returns the statistics about the kernel's use of virtual memory since the kernel was booted. The `pagesize` can also be found as a global variable `vm_page_size` which is set at task initialization and remains constant for the life of the task.

#### Returns

- `KERN_SUCCESS`
vm_machine_attribute

#include <mach.h>

kern_return_t vm_machine_attribute (task, address, size, attribute, value)
    task_t task;
    vm_address_t address;
    vm_size_t size;
    vm_machine_attribute_t attribute;
    vm_machine_attribute_val_t *value;

Arguments

task  The task whose memory is to be affected
address  Starting address of the memory segment.
size  Size of the memory segment
attribute  Attribute type
value  Pointer to the attribute's value

Description

vm_machine_attribute specifies machine-specific attributes for a VM mapping, such as cachability, migrability, replicability. This is used on machines that allow the user control over the cache (this is the case for MIPS architectures) or placement of memory pages as in NUMA architectures (Non-Uniform Memory Access time) such as the IBM ACE multiprocessor.

Machine-specific attributes can be consider additions to the machine-independent ones such as protection and inheritance, but they are not guaranteed to be supported by any given machine. Moreover, implementations of Mach on new architectures might find the need for new attribute types and or values besides the ones defined in the initial implementation.

The types currently defined are

MATTR_CACHE Controls caching of memory pages

MATTR_MIGRATE Controls migrability of memory pages

MATTR_REPLICATE Controls replication of memory pages

Corresponding values, and meaning of a specific call to vm_machine_attribute

MATTR_VAL_ON Enables the attribute. Being enabled is the default value for any applicable attribute.

MATTR_VAL_OFF Disables the attribute, making memory non-cached, or non-migratable, or non-replicatable.

MATTR_VAL_GET Returns the current value of the attribute for the memory segment. If the attribute does not apply uniformly to the given range the value returned applies to the initial portion of the segment only.

MATTR_VAL_CACHE_FLUSH Flush the memory pages from the Cache. The size value in this case
might be meaningful even if not a multiple of the page size, depending on the implementation.

MATTR_VAL_ICACHE_FLUSH Same as above, applied to the Instruction Cache alone.

MATTR_VAL_DCACHE_FLUSH Same as above, applied to the Data Cache alone.

Returns

KERN_SUCCESS The call succeeded.

KERN_INVALID_ARGUMENT task is not a task, or address and size do not define a valid address range in task, or attribute is not a valid attribute type, or it is not implemented, or value is not a permissible value for attribute.

Notes
The initial implementation (for MIPS) does not provide for inheritance of machine attributes. This might change if/when the IBM ACE code will be merged in the mainline.
6. Ancillary primitives
mach_ports
#include <mach.h>

kern_return_t mach_ports_register(target_task,
init_port_set, init_port_array_count)
    task_t    target_task;
    port_array_t init_port_set; /* array */
    int init_port_array_count;

kern_return_t mach_ports_lookup(target_task,
init_port_set, init_port_array_count)
    task_t    target_task;
    port_array_t *init_port_set; /* out array */
    int *init_port_array_count; /* out */

Arguments
    target_task      Task to be affected.
    init_port_set    An array of system ports to be registered, or returned. Although the array
                     size is given as variable, the MACH kernel will only accept a limited number
                     of ports.
    init_port_array_count
                     The number of ports returned in init_port_set.

Description
    mach_ports_register registers an array of well-known system ports with the kernel on behalf of a
specific task. Currently the ports to be registered are: the port to the Network Name Server, the port to
the Environment Manager, and a port to the Service server. These port values must be placed in specific
slots in the init_port_set. The slot numbers are given by the global constants defined in
mach_init.h: NAME_SERVER_SLOT, ENVIRONMENT_SLOT, and SERVICE_SLOT. These ports may
later be retrieved with mach_ports_lookup.

    When a new task is created (see task_create), the child task will be given access to these ports.
Only port send rights may be registered. Furthermore, the number of ports which may be registered is
fixed and given by the global constant MACH_PORT_SLOTS_USED. Attempts to register too many ports
will fail.

    It is intended that this mechanism be used only for task initialization, and then only by runtime support
modules. A parent task has three choices in passing these system ports to a child task. Most commonly it
can do nothing and its child will inherit access to the same init_port_set that the parent has; or a
parent task may register a set of ports it wishes to have passed to all of its children by calling
mach_ports_register using its task port; or it may make necessary modifications to the set of ports it
wishes its child to see, and then register those ports using the child’s task port prior to starting the child’s
thread(s). The mach_ports_lookup call which is done by mach_init in the child task will acquire
these initial ports for the child.

    Tasks other than the Network Name Server and the Environment Mangager 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.
Since the number of ports which may be registered is limited, ports other than those used by the runtime system to initialize a task should be passed to children either through an initial message, or through the Network Name Server for public ports, or the Environment Manager for private ports.

**Returns**

- **KERN_SUCCESS**: Memory allocated.
- **KERN_INVALID_ARGUMENT**: An attempt was made to register more ports than the current kernel implementation allows.

**See Also**

mach_init, netname, env_mgr, service
host_ipc_statistics
#include <mach.h>

kern_return_t host_ipc_statistics(task, statistics)
    task_t target_task;
    ipc_statistics_t *statistics; /* inout */

Arguments
    task Task running on the kernel whose statistics are desired.
    statistics The returned statistics.

Description
    host_ipc_statistics returns the statistics about MACH IPC, since the kernel was booted. statistics is a fixed length array provided by the user. See <kern/ipc_statistics.h> for a description of what is returned.

Returns
    KERN_SUCCESS The call succeeded.

Notes
    Only kernels compiled with MACH_IPCSTATS enabled support this call.

    The first argument should be a host port of some kind.

    The meaning of the statistics varies; not all fields are used.
7. External memory management primitives

7.1. Memory Managers

The MACH kernel allows users to provide memory management (i.e. paging) services outside the kernel. A server that provides such functions is called a memory manager. There is a default memory manager that is part of the kernel and is normally used to handle paging to both files and temporary memory objects. Users may provide additional memory managers to handle special kinds of objects, such as fault-tolerant objects, objects whose backing store is across a network link, or objects whose backing store is on devices for which the kernel does not provide drivers.

The protocol defined in this section consists of messages that the kernel will send to memory managers and the primitives that the kernel provides for the use of memory managers. Use of these primitives involves increased responsibility. A memory manager is expected to respond in a timely fashion to all the requests that the kernel makes of it, otherwise threads within the kernel are left hanging and the client task that is attempting to reference the memory object is also left hanging.

It is also possible for a privileged user to replace the default memory manager. This involves increased reliability and responsibility as now all the users of the system will be dependent on the new server.

7.1.1. Memory objects: definitions and basics

In MACH, physical memory is used as a cache of the contents of secondary storage objects called memory objects. The virtual address space of a task is represented as a series of mappings from contiguous virtual address ranges to such memory objects. For each memory object the kernel keeps track of those pages that are currently in the physical memory cache and it allows tasks mapped to that memory to use those physical pages.

When a virtual memory request occurs that cannot be resolved through the use of a previously cached physical page, the kernel must make a request of the memory object for the required data. As the physical page cache becomes full, the kernel must replace pages from the cache, writing the contents of modified pages back to the corresponding memory objects.

When a task uses the `vm_allocate` call, the kernel allocates a memory object that provides zero-filled memory on reference; this memory object is managed by a default memory manager.

Alternatively, a task may map a specific memory object into its address space by issuing a `vm_map` call. Included in this call is the memory object, represented by a port, that is to manage the data in the allocated region. The kernel will use the memory object port to make requests for data, or to request that data be written back to the object. The memory manager must act as a server for these requests. The memory manager server interface differs from other servers only in that the kernel does not synchronously await replies.

A given memory object may be mapped into an arbitrary number of tasks, at any addresses available in those tasks. When a `vm_map` call is issued, the MACH kernel will recognize the memory object if it has been mapped before; any physical memory pages from this memory object already cached from previous uses may be shared by later mappings as well. A single MACH kernel keeps the physical memory cache consistent across all uses of the same memory object at similar page alignments on that host.
Furthermore, a single memory object may be mapped into tasks created on different hosts (and therefore be cached by different MACH kernels). In this case, the memory manager is responsible for maintaining any desired consistency among the various hosts on which its data resides.

7.1.2. Initialization and termination

The memory manager must define a protocol for giving out memory object ports. This could take the form of the memory manager registering a general service port somewhere that clients could find and exporting an object create or object lookup call that will return a memory object port. This is the port that is passed to the kernel in the `vm_map` call.

Upon processing the first `vm_map` call for a given memory object, the MACH kernel will make a `memory_object_init` call, providing the memory manager with two ports: a control port, and a name port. The memory manager may use the memory object control port to supply the kernel with data for it to cache, or to perform other cache management functions. These requests will be covered in the next section.

The memory object name, a port, will only be used by the kernel in the results from a `vm_region` call to describe the source of data for a given region. Since this port is not to be used for requests for data, the memory manager may wish to provide this port to clients to identify memory which it supplies.

The initialization call also includes the system page size for the host on which the mapping took place. This allows the memory manager to provide data to the kernel in whole pages, and to detect mappings at inconsistent page alignments.

In order to indicate its readiness to accept requests, the memory manager must respond to the initialization call by making a `memory_object_set_attributes` call, asserting the readiness parameter.

Normally, when a memory object is no longer referenced by any virtual address space, the MACH kernel will deallocate its port rights to that memory object after sending all port rights for the control and name ports in an `memory_object_terminate` call. To enhance performance, a memory manager may allow a MACH kernel to maintain its memory cache for a memory object after all virtual address space references to it are gone, by asserting the caching parameter to the `memory_object_set_attributes` call. However, allowing caching does not prevent the kernel from terminating an object.

In the event that a memory manager destroys a memory object port that is currently mapped into one or more virtual address spaces, future page faults on addresses mapped to this object (for which data is not available in the cache) will result in a memory exception.

7.1.3. Kernel-created memory objects

As noted earlier, memory created using `vm_allocate` results in the creation of a memory object; this object is created by the kernel, and is passed to the default memory manager, using the `memory_object_create` call. Since the memory object is initially zero-filled, it only contains data that has been modified.

The `memory_object_create` request will only be made of the default memory manager. The default
memory manager must not allow any memory object passed in a `memory_object_create` call to be used in any other task, as the kernel may make assumptions about such an object that could adversely affect external consistency.

7.2. Kernel calls supporting memory managers
vm_map
#include <mach.h>

kern_return_t vm_map(target_task, address, size, mask, anywhere,
                      memory_object, offset, copy,
                      cur_protection, max_protection,
                      inheritance)

    task_t    target_task;
    vm_offset_t    *address;    /* in/out */
    vm_size_t    size;
    vm_offset_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

    vm_map maps a region of virtual memory at the specified address, for which data is to be supplied by
the given memory object, starting at the given offset within that object. In addition to the arguments used
in vm_allocate, the vm_map call allows the specification of an address alignment parameter, and of the
initial protection and inheritance values. [See the descriptions of vm_allocate, vm_protect, and
vm_inherit.]

    If the memory object in question is not currently in use, the MACH kernel will perform a
memory_object_init call at this time. If the copy parameter is asserted, the specified region of the
memory object will be copied to this address space; changes made to this object by other tasks will not be
visible in this mapping, and changes made in this mapping will not be visible to others (or returned to the
memory object).

    The vm_map call returns once the mapping is established. Completion of the call does not require any
action on the part of the memory manager.

Warning: Only memory objects that are provided by bona fide memory managers should be used in
the vm_map call. A memory manager must implement the memory object interface described elsewhere
in this manual. If other ports are used, a thread that accesses the mapped virtual memory may become
permanently hung or may receive a memory exception.

Arguments

    target_task        Task to be affected.
    address            Starting address. If the anywhere option is used, this address is ignored.
The address actually allocated will be returned in address.
    size               Number of bytes to allocate (rounded by the system in a machine dependent
                        way).
    mask               Alignment restriction. Bits asserted in this mask must not be asserted in the
                        address returned.
    anywhere           If set, the kernel should find and allocate any region of the specified size, and
                        return the address of the resulting region in address.
memory_object Port that represents the memory object: used by user tasks in vm_map; used by the MACH kernel to make requests for data or other management actions. If this port is MEMORY_OBJECT_NULL, then zero-filled memory is allocated instead.

offset An offset within a memory object, in bytes. This must be page aligned.

copy If set, the range of the memory object should be copied to the target task, rather than mapped read-write.

Returns

KERN_SUCCESS The object is mapped.

KERN_NO_SPACE No unused region of the task’s virtual address space that meets the address, size, and alignment criteria could be found.

KERN_INVALID_ARGUMENT An illegal argument was provided.

See Also

memory_object_server, vm_allocate
memory_object_set_attributes

#include <mach.h>

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

Description

memory_object_set_attributes controls how the MACH kernel uses the memory object. The
kernel will only make data or unlock requests when the ready attribute is asserted. If the caching attribute
is asserted, the kernel is permitted (and encouraged) to maintain cached data for this memory object
even after no virtual address space contains this data.

There are three possible caching strategies: MEMORY_OBJECT_COPY_NONE which specifies that
nothing special should be done when data in the object is copied; MEMORY_OBJECT_COPY_CALL which
specifies that the memory manager should be notified via a memory_object_copy call before any part
of the object is copied; and MEMORY_OBJECT_COPY_DELAY which guarantees that the memory manager
does not externally modify the data so that the kernel can use its normal copy-on-write algorithms.
MEMORY_OBJECT_COPY_DELAY is the strategy most commonly used.

Arguments

memory_control The port, provided by the kernel in a memory_object_init call, to which
              cache management requests may be issued.

object_ready When set, the kernel may issue new data and unlock requests on the
              associated memory object.

may_cache_object If set, the kernel may keep data associated with this memory object, even
                  after virtual memory references to it are gone.

copy_strategy How the kernel should copy regions of the associated memory object.

Returns

KERN_SUCCESS This routine does not receive a reply message (and consequently has no
                return value), so only message transmission errors apply.

See Also

memory_object_init, memory_object_copy, memory_object_attributes
memory_object_get_attributes
#include <mach.h>

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

Description
memory_object_get_attributes retrieves the current attributes associated with the memory object.

Arguments
memory_control The port, provided by the kernel in a memory_object_init call, to which cache management requests may be issued.
object_ready When set, the kernel may issue new data and unlock requests on the associated memory object.
may_cache_object If set, the kernel may keep data associated with this memory object, even after virtual memory references to it are gone.
copy_strategy How the kernel should copy regions of the associated memory object.

Returns
KERN_SUCCESS This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

See Also
memory_object_set_attributes, memory_object_copy
memory_object_lock_request
#include <mach.h>

kern_return_t memory_object_lock_request(memory_control,
    offset, size, should_clean
    should_flush, lock_value, reply_to)

    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;
    port_t reply_to;

Description
memory_object_lock_request allows a memory manager to make cache management requests. As specified in arguments to the call, the kernel will: clean (i.e., write back using memory_object_data_write) any cached data which has been modified since the last time it was written; flush (i.e., remove any uses of) that data from memory; lock (i.e., prohibit the specified uses of) the cached data. Locks applied to cached data are not cumulative; new lock values override previous ones. Thus, data may also be unlocked using this primitive. The lock values must be one or more of the following values: VM_PROT_NONE, VM_PROT_READ, VM_PROT_WRITE, VM_PROT_EXECUTE and VM_PROT_ALL as defined in <mach/vm_prot.h>.

Only data which is cached at the time of this call is affected. When a running thread requires a prohibited access to cached data, the MACH kernel will issue a memory_object_data_unlock call specifying the forms of access required. Once all of the actions requested by this call have been completed, the MACH kernel will issue a memory_object_lock_completed call on the specified reply port.

Arguments
memory_control The port, provided by the kernel in a memory_object_init call, to which cache management requests may be issued.
offset An offset within a memory object, in bytes. This must be page aligned.
size The amount of cached data (starting at offset) to be handled, must be an integral multiple of the memory object page size.
should_clean If set, modified data should be written back to the memory manager.
should_flush If set, the specified cached data should be invalidated, and all uses of that data should be revoked.
lock_value A protection value indicating those forms of access that should not be permitted to the specified cached data.
reply_to A port on which a memory_object_lock_completed call should be issued, or PORT_NULL if no acknowledgement is desired.

Returns
KERN_SUCCESS This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.
See Also
memory_object_lock_completed, memory_object_data_unlock
memory_object_data_provided
#include <mach.h>

kern_return_t memory_object_data_provided(memory_control,
                                        offset, data, data_count, lock_value)

    memory_object_control_t memory_control;
    vm_offset_t offset;
    pointer_t data;
    int data_count;
    vm_prot_t lock_value;

Description
memory_object_data_provided supplies the kernel with data for the specified memory object. Ordinarily, memory managers should only provide data in response to memory_object_data_request calls from the kernel. The lock_value specifies what type of access will not be allowed to the data range. The lock values must be one or more of the set: VM_PROT_NONE, VM_PROT_READ, VM_PROT_WRITE, VM_PROT_EXECUTE and VM_PROT_ALL as defined in <mach/vm_prot.h>.

Arguments
memory_control The port, provided by the kernel in a memory_object_init call, to which cache management requests may be issued.
offset An offset within a memory object, in bytes. This must be page aligned.
data Data that is being provided to the kernel. This is a pointer to the data.
data_count The amount of data to be provided. Must be an integral number of memory object pages.
lock_value A protection value indicating those forms of access that should not be permitted to the specified cached data.

Returns
KERN_SUCCESS This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

See Also
memory_object_data_request, memory_object_data_error, memory_object_lock_request
memory_object_data_unavailable
#include <mach.h>

kern_return_t memory_object_data_unavailable(memory_control,
       offset, size);

   memory_object_control_t
       memory_control;
   vm_offset_t    offset;
   vm_size_t      size;

Description
memory_object_data_unavailable indicates that the memory object does not have data for the
given region and that the kernel should provide the data for this range. The memory manager may use
this call in three different situations. 1) The object was created by memory_object_create and the
kernel has not yet provided data for this range (either via a memory_object_data_initialize or a
memory_object_data_write. In this case the kernel should supply zero-filled pages for the object. 2)
The object was created by an memory_object_data_copy and the kernel should copy this region from
the original memory object. 3) The object is a normal user-created memory object and the kernel should
supply unlocked zero-filled pages for the range.

Arguments
   memory_control  The port, provided by the kernel in a memory_object_init call, to which
                   cache management requests may be issued.
   offset          An offset within a memory object, in bytes.  This must be page aligned.
   size            The amount of cached data (starting at offset) to be handled. This must be
                   an integral multiple of the memory object page size.

Returns
   KERN_SUCCESS    This routine does not receive a reply message (and consequently has no
                   return value), so only message transmission errors apply.

See Also
   memory_object_create, memory_object_data_request, memory_object_data_error
memory_object_data_error
#include <mach.h>

kern_return_t memory_object_data_error(memory_control, offset, size, reason);

memory_object_control_t memory_control;
vm_offset_t offset;
vm_size_t size;
kern_return_t reason;

Description
memory_object_data_error indicates that the memory manager cannot return the data requested for the given region, specifying a reason for the error. This is typically used when a hardware error is encountered.

Arguments
memory_control The port, provided by the kernel in a memory_object_init call, to which cache management requests may be issued.
offset An offset within a memory object, in bytes. This must be page aligned.
size The amount of cached data (starting at offset) to be handled. This must be an integral multiple of the memory object page size.
reason Could be a Unix error code for a hardware error.

Returns
KERN_SUCCESS This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

See Also
memory_object_data_request, memory_object_data_provided

Notes
The error code is currently ignored.
**memory_object_destroy**

```c
#include <mach.h>

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

### Description

`memory_object_destroy` tells the kernel to shut down the memory object. As a result of this call the kernel will no longer support paging activity or any `memory_object` calls on this object, and all rights to the memory object port, the memory control port and the memory name port will be returned to the memory manager in a `memory_object_terminate` call. If the memory manager is concerned that any modified cached data be returned to it before the object is terminated, it should call `memory_object_lock_request` with `should_flush` set and a lock value of `VM_PROT_WRITE` before making this call.

### Arguments

- **memory_control** The port, provided by the kernel in a `memory_object_init` call, to which cache management requests may be issued.
- **reason** An error code indicating when the object must be destroyed.

### Returns

- **KERN_SUCCESS** This routine does not receive a reply message (and consequently has no return value), so only message transmission errors apply.

### See Also

- `memory_object_terminate`, `memory_object_lock_request`

### Notes

The error code is currently ignored.
vm_set_default_memory_manager
#include <mach.h>

routine vm_set_default_memory_manager(host,default_manager)
    task_t host;
    memory_object_t default_manager; /* in/out */

Description

vm_set_default_memory_manager sets the kernel’s default memory manager. It sets the port to
which newly-created temporary memory objects are delivered by memory_object_create to the host.
The old memory manager port is returned. If default_manager is PORT_NULL then this routine just
returns the current default manager port without changing it.

Arguments

    host  A task port to the kernel whose default memory manager is to be changed.
    default_manager
        Input as the port that the new memory manager is listening on for
        memory_object_create calls. Returned as the old default memory
        manager’s port.

Returns

    KERN_SUCCESS  The new memory manager is installed.
    KERN_INVALID_ARGUMENT  This task does not have the privileges required for this call.

See Also

    vm_allocate, memory_object_create, memory_object_data_initialize

Notes

    There is no way for the user task to acquire the appropriate privilege to make this call.

<<<
7.3. Memory Manager calls

This section describes calls made by the MACH kernel on a memory object that has previously been mapped by some task (see `vm_map`). A task that manages a memory object (called a **memory manager**) must act as a server for this interface.

In order to isolate the memory manager from the specifics of message formatting, the remote procedure call generator, MIG, produces a procedure, `memory_object_server`, to handle a received message. This function does all necessary argument handling, and calls one of the interface functions described below.

The procedures described in this section are the calls that the kernel may make to a memory manager either as a result of a user action on a memory object or as part of the kernel’s physical memory management. To be useful a memory manager must define at least a couple more protocols. It must make a service port available to potential clients and it must provide a way for clients to get a memory object port to hand to the `vm_map` call. It may also wish to provide calls for clients to get or pass information about a specific memory object. The memory object name port can be used for this purpose.

The kernel includes a default memory manager which handles those memory objects that it needs to create or are created by a user with the call `vm_allocate`. The user may substitute a new default memory manager if he wishes with the privileged call `vm_set_default_memory_manager`. The final two calls in the section are only made to the default memory manager. Other memory managers need not provide these calls.

These calls are the result of an asynchronous message sent by the kernel, i.e., the kernel does not wait for a reply to the message. Thus the error returned from these calls are ignored; however, most require some action on the part of the memory manager. These response actions need not necessarily be done in the order requested, but should be done as soon as practical.

The calls that are made by the kernel to all memory managers are:

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

The following two calls must also be provided by the default memory manager.

- `memory_object_create`
- `memory_object_data_initialize`
memory_object_server
#include <mach.h>

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

Description

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.

In order to isolate the memory manager from the specifics of message formatting, the remote procedure call generator produces a procedure, memory_object_server, to handle a received message. This function does all necessary argument handling, and actually calls one of the following functions: 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. A default memory manager may get two additional requests from the kernel: memory_object_create and memory_object_data_initialize.

The return value from the memory_object_server function indicates that the message was appropriate to the memory management interface (returning TRUE), or that it could not handle this message (returning FALSE).

Arguments

    in_msg The message that has been received from the kernel.
    out_msg A reply message. Not used for this server

Returns

    TRUE From memory_object_server, indicates that the message in question was applicable to this interface, and that the appropriate routine was called to interpret the message.
    FALSE From memory_object_server, indicates that the message did not apply to this interface, and that no other action was taken.

See Also

memory_object_init, memory_object_data_request, memory_object_data_unlock,
memory_object_data_write, memory_object_copy, memory_object_terminate,
memory_object_lock_completed, memory_object_data_initialize,
memory_object_create
memory_object_init
#include <mach.h>

kern_return_t memory_object_init(memory_object, memory_control,
                                  memory_object_name, memory_object_page_size)
    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
memory_object_init serves as a notification that a MACH kernel has been asked to map the given
memory object into a task's virtual address space. Additionally, it provides a port on which the memory
manager may issue cache management requests, and a port which the kernel will use to name this data
region. In the event that different MACH kernels are asked to map the same memory object, each will
perform a memory_object_init call with new request and name ports. The virtual page size that is
used by the calling kernel is included for planning purposes.

When the memory manager is prepared to accept requests for data for this object, it should call
memory_object_set_attribute with the attribute ready set. Otherwise the kernel will not process
requests on this object.

Arguments
    memory_object The port that represents the memory object data, as supplied to the kernel in
                    a vm_map call.
    memory_control The request port to which a response is requested. [In the event that a
                    memory object has been supplied to more than one MACH kernel, this
                    argument identifies the kernel that has made the request.]
    memory_object_name A 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 to be used by this kernel. All data sizes in calls involving this
                             kernel must be an integral multiple of the page size. [Note that different
                             kernels, indicated by different memory_controls may have different page
                             sizes.]

Returns
    KERN_SUCCESS Since this routine is called by the kernel, which does not wait for a reply
                     message, this value is ignored.

See Also
    memory_object_set_attributes
memory_object_data_request

#include <mach.h>

kern_return_t memory_object_data_request(memory_object, memory_control, offset, length, desired_access)

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

memory_object_data_request is a request for data from the specified memory object, for at least the access specified. The memory manager is expected to return at least the specified data, with as much access as it can allow, using memory_object_data_provided. If the memory manager is unable to provide the data (for example, because of a hardware error), it may use the memory_object_data_error call. memory_object_data_unavailable call may be used to tell the kernel to supply zero-filled memory for this region.

Arguments

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

memory_control The request port to which a response is requested. [In the event that a memory object has been supplied to more than one MACH kernel, this argument identifies the kernel that has made the request.]

offset The offset within a memory object to which this call refers. This will be page aligned.

length The number of bytes of data, starting at offset, to which this call refers. This will be an integral number of memory object pages.

desired_access A protection value describing the memory access modes which must be permitted on the specified cached data. One or more of: VM_PROT_READ, VM_PROT_WRITE or VM_PROT_EXECUTE.

Returns

KERN_SUCCESS Since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

See Also

memory_object_data_provided, memory_object_data_error, memory_object_data_unavailable
memory_object_data_write

#include <mach.h>

kern_return_t memory_object_data_write(memory_object, memory_control, offset, data, data_count)

memory_object_t memory_object;
memory_object_control_t memory_control;
vm_offset_t offset;
pointer_t data;
unsigned int data_count;

Description

memory_object_data_write provides the memory manager with data that has been modified while cached in physical memory. Once the memory manager no longer needs this data (e.g., it has been written to another storage medium), it should be deallocated using vm_deallocate.

Arguments

memory_object The port that represents the memory object data, as supplied to the kernel in a vm_map call.
memory_control The request port to which a response is requested. [In the event that a memory object has been supplied to more than one MACH kernel, this argument identifies the kernel that has made the request.]
offset The offset within a memory object to which this call refers. This will be page aligned.
data Data which has been modified while cached in physical memory.
data_count The amount of data to be written, in bytes. This will be an integral number of memory object pages.

Returns

KERN_SUCCESS Since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

See Also

vm_deallocate
memory_object_data_unlock
#include <mach.h>

kern_return_t memory_object_data_unlock(memory_object, memory_control,
    offset, length, desired_access)

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
memory_object_data_unlock is a request that the memory manager permit at least the desired
access to the specified data cached by the kernel. A call to memory_object_lock_request is
expected in response.

Arguments

memory_object  The port that represents the memory object data, as supplied to the kernel in
    a vm_map call.
memory_control The request port to which a response is requested. [In the event that a
    memory object has been supplied to more than one MACH kernel, this
    argument identifies the kernel that has made the request.]
offset          The offset within a memory object to which this call refers. This will be page
    aligned.
length          The number of bytes of data, starting at offset, to which this call refers.
    This will be an integral number of memory object pages.
desired_access  A protection value describing the memory access modes which must be
    permitted on the specified cached data. One or more of: VM_PROT_READ,
    VM_PROT_WRITE or VM_PROT_EXECUTE.

Returns

KERN_SUCCESS   Since this routine is called by the kernel, which does not wait for a reply
    message, this value is ignored.

See Also
memory_object_lock_request, memory_object_lock_completed
memory_object_copy
#include <mach.h>

kern_return_t memory_object_copy(old_memory_object, old_memory_control,
                                 offset, length, new_memory_object)

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
memory_object_copy indicates that a copy has been made of the specified range of the given original memory object. This call includes only the new memory object itself; a memory_object_init call will be made on the new memory object after the currently cached pages of the original object are prepared. After the memory manager receives the init call, it should reply with the memory_object_set_attributes call to assert the "ready" attribute. The kernel will use the new memory object, control and name ports to refer to the new copy.

This call is made when the original memory object had the caching parameter set to MEMORY_OBJECT_COPY_CALL and a user of the object has asked the kernel to copy it.

Cached pages from the original memory object at the time of the copy operation are handled as follows: Readable pages may be silently copied to the new memory object (with all access permissions). Pages not copied are locked to prevent write access.

The new memory object is temporary, meaning that the memory manager should not change its contents or allow the memory object to be mapped in another client. The memory manager may use the memory_object_data_unavailable call to indicate that the appropriate pages of the original memory object may be used to fulfill the data request.

Arguments

old_memory_object
The port that represents the old memory object data.

old_memory_control
The kernel control port for the old object.

offset
The offset within a memory object to which this call refers. This will be page aligned.

length
The number of bytes of data, starting at offset, to which this call refers. This will be an integral number of memory object pages.

new_memory_object
A new memory object created by the kernel; see synopsis for further description. Note that all port rights (including receive rights) are included for the new memory object.
Returns

KERN_SUCCESS   Since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

See Also

memory_object_init,
memory_object_set_attributes,
memory_object_data_unavailable
memory_object_terminate

#include <mach.h>

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

Description

memory_object_terminate indicates that the MACH kernel has completed its use of the given memory object. All rights to the memory object control and name ports are included, so that the memory manager can destroy them (using port_deallocate) after doing appropriate bookkeeping. The kernel will terminate a memory object only after all address space mappings of that memory object have been deallocated, or upon explicit request by the memory manager.

Arguments

memory_object The port that represents the memory object data, as supplied to the kernel in a vm_map call.
memory_control The request port to which a response is requested. [In the event that a memory object has been supplied to more than one MACH kernel, this argument identifies the kernel that has made the request.]
memory_object_name A port used by the kernel to refer to the memory object data in response to vm_region calls.

Returns

KERN_SUCCESS Since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

See Also

memory_object_destroy, port_deallocate
memory_object_create

#include <mach.h>

kern_return_t memory_object_create(old_memory_object, new_memory_object,
new_object_size, new_control,
new_name, new_page_size)

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

memory_object_create is a request that the given memory manager accept responsibility for the
given memory object created by the kernel. This call will only be made to the system default memory
manager. The memory object in question initially consists of zero-filled memory; only memory pages that
are actually written will ever be provided to the memory manager. When processing
memory_object_data_request calls, the default memory manager must use
memory_object_data_unavailable for any pages that have not previously been written.

No reply is expected after this call. Since this call is directed to the default memory manager, the kernel
assumes that it will be ready to handle data requests to this object and does not need the confirmation of
a memory_object_set_attributes call.

Arguments

old_memory_object
A memory object provided by the default memory manager on which the
kernel can make memory_object_create calls.

new_memory_object
A new memory object created by the kernel; see synopsis for further
description. Note that all port rights (including receive rights) are included for
the new memory object.

new_object_size
Maximum size of the new object.

new_control
A port, created by the kernel, on which a memory manager may issue cache
management requests for the new object.

new_name
A 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 to be used by this kernel. All data sizes in calls involving this
kernel must be an integral multiple of the page size. [Note that different
kernels, indicated by different memory_controls may have different page
sizes.]
Returns

KERN_SUCCESS  Since this routine is called by the kernel, which does not wait for a reply message, this value is ignored.

See Also

memory_object_data_initialize
memory_object_data_initialize

#include <mach.h>

kern_return_t memory_object_data_initialize(memory_object, memory_control, 
                                           offset, data, data_count)

memory_object_t memory_object;
memory_object_control_t
    memory_control;
vm_offset_t offset;
pointer_t data;
unsigned int data_count;

Description

memory_object_data_initialize provides the memory manager with initial data for a kernel-
created memory object. If the memory manager already has been supplied data (by a previous
memory_object_data_initialize or memory_object_data_write), then this data should be
ignored. Otherwise, this call behaves exactly as does memory_object_data_write. This call will only
be made on memory objects created by the kernel via memory_object_create and thus will only be
made to default memory managers. This call will not be made on objects created via
memory_object_copy.

Arguments

memory_object The port that represents the memory object data, as supplied by the kernel in
               a memory_object_create call.
memory_control The request port to which a response is requested. [In the event that a
               memory object has been supplied to more than one MACH kernel, this
               argument identifies the kernel that has made the request.]
offset The offset within a memory object to which this call refers. This will be page
         aligned.
data Data which has been modified while cached in physical memory.
data_count The amount of data to be written, in bytes. This will be an integral number of
             memory object pages.

Returns

KERN_SUCCESS Since this routine is called by the kernel, which does not wait for a reply
message, this value is ignored.

See Also

memory_object_data_write, memory_object_create
I. Summary of Kernel Calls

The following is a summary of calls to the MACH kernel. The page on which the operation is fully described appears within square brackets.

[7] msg_return_t msg_send(header, option, timeout)
    msg_header_t *header;
    msg_option_t option;
    msg_timeout_t timeout;

[9] msg_return_t msg_receive(header, option, timeout)
    msg_header_t *header; /* in/out */
    msg_option_t option;
    msg_timeout_t timeout;

[11] msg_return_t msg_rpc(header, option, rcv_size,
    send_timeout, rcv_timeout)
    msg_header_t *header; /* in/out */
    msg_option_t option;
    msg_size_t rcv_size;
    msg_timeout_t send_timeout;
    msg_timeout_t rcv_timeout;

[13] kern_return_t port_names(task,
    portnames, portnamesCnt,
    port_types, port_typesCnt)
    task_t task;
    port_name_array_t *portnames; /* out array */
    unsigned int *portnamesCnt; /* out */
    port_type_array_t *port_types; /* out array */
    unsigned int *port_typesCnt; /* out */

[14] kern_return_t port_type(task, port_name, port_type)
    task_t task;
    port_name_t port_name;
    port_type_t *port_type; /* out */

[15] kern_return_t port_rename(task, old_name, new_name)
    task_t task;
    port_name_t old_name;
    port_name_t new_name;

[16] kern_return_t port_allocate(task, port_name)
    task_t task;
port_name_t *port_name; /* out */

[17] kern_return_t port_deallocate(task, port_name)
    task_t task;
    port_name_t port_name;

[18] kern_return_t port_status(task, port_name, enabled, num_msgs, backlog, owner, receiver)
    task_t task;
    port_name_t port_name;
    port_set_name_t *enabled; /* out */
    int *num_msgs; /* out */
    int *backlog; /* out */
    boolean_t *owner; /* out */
    boolean_t *receiver; /* out */

[19] kern_return_t port_set_backlog(task, port_name, backlog)
    task_t task;
    port_name_t port_name;
    int backlog;

[20] kern_return_t port_set_backup(task, primary, backup, previous)
    task_t task;
    port_name_t primary;
    port_t backup;
    port_t *previous; /* out */

[21] kern_return_t port_set_allocate(task, set_name)
    task_t task;
    port_set_name_t *set_name; /* out */

[22] kern_return_t port_set_deallocate(task, set_name)
    task_t task;
    port_set_name_t set_name;

[23] kern_return_t port_set_add(task, set_name, port_name)
    task_t task;
    port_set_name_t set_name;
    port_name_t port_name;
[24] kern_return_t port_set_remove(task, port_name)
    task_t task;
    port_name_t port_name;

[25] kern_return_t port_set_status(task, set_name, members, membersCnt)
    task_t task;
    port_set_name_t set_name;
    port_name_array_t *members; /* out array */
    unsigned int *membersCnt; /* out */

[26] kern_return_t port_insert_send(task, my_port, his_name)
    task_t task;
    port_t my_port;
    port_name_t his_name;

[26] kern_return_t port_insert_receive(task, my_port, his_name)
    task_t task;
    port_t my_port;
    port_name_t his_name;

[27] kern_return_t port_extract_send(task, his_name, his_port)
    task_t task;
    port_name_t his_name;
    port_t *his_port; /* out */

[27] kern_return_t port_extract_receive(task, his_name, his_port)
    task_t task;
    port_name_t his_name;
    port_t *his_port; /* out */

[30] kern_return_t task_create(parent_task, inherit_memory, child_task)
    task_t parent_task
    boolean_t inherit_memory;
    task_t *child_task; /* out */

[31] kern_return_t task_terminate(target_task)
    task_t target_task;
kern_return_t task_suspend(target_task)
    task_t target_task;

kern_return_t task_resume(target_task)
    task_t target_task;

kern_return_t task_get_special_port(task, which_port, special_port)
    task_t task;
    int which_port;
    port_t *special_port; /* out */

kern_return_t task_set_special_port(task, which_port, special_port)
    task_t task;
    int which_port;
    port_t special_port;

task_t task_self()

port_t task_notify()

kern_return_t task_info(target_task, flavor, task_info, task_infoCnt)
    task_t target_task;
    int flavor;
    task_info_t task_info; /* in and out */
    unsigned int *task_infoCnt; /* in and out */

kern_return_t task_threads(target_task, thread_list, thread_count)
    task_t target_task;
    thread_array_t *thread_list; /* out, ptr to array */
    int *thread_count; /* out */

kern_return_t thread_create(parent_task, child_thread)
    task_t parent_task;
    thread_t *child_thread; /* out */
kern_return_t thread_terminate(target_thread)
    thread_t target_thread;

kern_return_t thread_suspend(target_thread);
    thread_t target_thread;

kern_return_t thread_resume(target_thread)
    thread_t target_thread;

kern_return_t thread_abort(target_thread)
    thread_t target_thread;

kern_return_t thread_get_special_port(thread, which_port, special_port)
    thread_t thread;
    int which_port;
    port_t *special_port;

kern_return_t thread_set_special_port(thread, which_port, special_port)
    thread_t thread;
    int which_port;
    port_t special_port;

thread_t thread_self()

port_t thread_reply()

kern_return_t thread_info(target_thread, flavor, thread_info, thread_infoCnt)
    thread_t target_thread;
    int flavor;
    thread_info_t thread_info; /* in and out */
    unsigned int *thread_infoCnt; /* in and out */

kern_return_t thread_get_state(target_thread, flavor, old_state, old_stateCnt)
    thread_t target_thread;
int flavor;
thread_state_data_t old_state; /* in and out */
unsigned int *old_stateCnt; /* in and out */

kern_return_t thread_set_state(target_thread, flavor, new_state,
new_stateCnt)
thread_t target_thread;
int flavor;
thread_state_data_t new_state;
unsigned int new_stateCnt;

vm_task_t target_task;
vm_address_t *address; /* in/out */
vm_size_t size;
boolean_t anywhere;

kern_return_t vm_allocate(target_task, address, size, anywhere)
vm_task_t target_task;
vm_address_t *address; /* in/out */
vm_size_t size;

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

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

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

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

kern_return_t vm_region(target_task, address, size, protection,
max_protection, inheritance, shared,
object_name, offset)

vm_task_t target_task;
vm_address_t *address; /* in/out */
vm_size_t *size; /* out */
vm_prot_t *protection; /* out */
vm_prot_t *max_protection; /* out */
vm_inherit_t *inheritance; /* out */
boolean_t *shared; /* out */
port_t *object_name; /* out */
vm_offset_t *offset; /* out */

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

[59] kern_return_t vm_inherit(target_task, address, size, new_inheritance)
    vm_task_t target_task;
    vm_address_t address;
    vm_size_t size;
    vm_inherit_t new_inheritance;

[60] kern_return_t vm_statistics(target_task, vm_stats)
    task_t target_task;
    vm_statistics_data_t *vm_stats; /* out */

[61] kern_return_t vm_machine_attribute (task, address, size, attribute, valu
    task_t task;
    vm_address_t address;
    vm_size_t size;
    vm_machine_attribute_t attribute;
    vm_machine_attribute_val_t *value;

[64] kern_return_t mach_ports_register(target_task,
    init_port_set, init_port_array_count)
    task_t target_task;
    port_array_t init_port_set; /* array */
    int init_port_array_count;

[64] kern_return_t mach_ports_lookup(target_task,
    init_port_set, init_port_array_count)
task_t target_task;
port_array_t *init_port_set; /* out array */
int *init_port_array_count; /* out */

[66] kern_return_t host_ipc_statistics(task, statistics)
    task_t target_task;
    ipc_statistics_t *statistics; /* inout */

[70] kern_return_t vm_map(target_task, address, size, mask, anywhere,
    memory_object, offset, copy, cur_protection, max_protection,
    inheritance)
    task_t target_task;
    vm_offset_t *address; /* in/out */
    vm_size_t size;
    vm_offset_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;

[72] kern_return_t memory_object_set_attributes(memory_control,
    object_ready, may_cache_object, copy_strategy)
    memory_object_control_t
        memory_control;
    boolean_t object_ready;
    boolean_t may_cache_object;
    memory_object_copy_strategy_t
        copy_strategy;

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

[74] kern_return_t memory_object_lock_request(memory_control,
    offset, size, should_clean
    should_flush, lock_value, reply_to)
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;
port_t reply_to;

[76] kern_return_t memory_object_data_provided(memory_control,
    offset, data, data_count, lock_value)

memory_object_control_t memory_control;
vm_offset_t offset;
pointer_t data;
int data_count;
vm_prot_t lock_value;

[77] kern_return_t memory_object_data_unavailable(memory_control,
    offset, size);

memory_object_control_t memory_control;
vm_offset_t offset;
vm_size_t size;

[78] kern_return_t memory_object_data_error(memory_control,
    offset, size, reason);

memory_object_control_t memory_control;
vm_offset_t offset;
vm_size_t size;
kern_return_t reason;

[79] kern_return_t memory_object_destroy(memory_control, reason);

memory_object_control_t memory_control;
kern_return_t reason;

[80] routine vm_set_default_memory_manager(host, default_manager)
task_t host;
memory_object_t default_manager; /* in/out */
II. Summary of External Memory Management Calls

The following is a summary of calls that the MACH kernel makes on an external memory management server. The page on which the operation is fully described appears within square brackets.

[82] boolean_t memory_object_server(in_msg, out_msg)
    msg_header_t *in_msg;
    msg_header_t *out_msg;

[83] kern_return_t memory_object_init(memory_object, memory_control, memory_object_name, memory_object_page_size)
    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;

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

[85] kern_return_t memory_object_data_write(memory_object, memory_control, offset, data, data_count)
    memory_object_t memory_object;
    memory_object_control_t memory_control;
    vm_offset_t offset;
    pointer_t data;
    unsigned int data_count;

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

[87] kern_return_t memory_object_copy(old_memory_object, old_memory_control, offset, length, new_memory_object)
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;

[89] kern_return_t memory_object_terminate(memory_object, memory_control, memory_object_name)
    memory_object_t memory_object;
    memory_object_control_t memory_control;
    memory_object_name_t memory_object_name;

[90] kern_return_t memory_object_create(old_memory_object, new_memory_object, new_object_size, new_control, new_name, new_page_size)
    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;

[92] kern_return_t memory_object_data_initialize(memory_object, memory_control, offset, data, data_count)
    memory_object_t memory_object;
    memory_object_control_t memory_control;
    vm_offset_t offset;
    pointer_t data;
    unsigned int data_count;
# Table of Contents

1. Introduction 1  
   1.1. Overall system organization 1  
   1.2. Basic kernel functionality 1  
   1.3. User operating system environments 2  

2. Message primitives 3  
   2.1. Basic terms 3  
   2.2. Ports 3  
   2.3. Port sets 4  
   2.4. Port names 4  
   2.5. Port types 4  
   2.6. Messages 5  

3. Port and port set primitives 12  

4. Task and thread primitives 28  
   4.1. Basic terms 28  
   4.2. Access to Tasks: Terminology 29  

5. Virtual memory primitives 51  
   5.1. Basic terms 51  

6. Ancillary primitives 63  

7. External memory management primitives 67  
   7.1. Memory Managers 67  
      7.1.1. Memory objects: definitions and basics 67  
      7.1.2. Initialization and termination 68  
      7.1.3. Kernel-created memory objects 68  
   7.2. Kernel calls supporting memory managers 69  
   7.3. Memory Manager calls 81  

I. Summary of Kernel Calls 93  

II. Summary of External Memory Management Calls 102