Mach 3 Server Writer’s Interfaces

Open Software Foundation and Carnegie Mellon University

Keith Loepere, Editor

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

Books in the OSF Mach 3 series:

Mach 3 Kernel Principles

Mach 3 Kernel Interfaces

Mach 3 Server Writer’s Guide

Mach 3 Server Writer’s Interfaces

Revision History:

Revision 2	MK67, user11:January 15, 1992	OSF Mach release
Revision 2.1	NORMA-MK12, user15: July 15, 1992	

Change bars indicate change since MK67, user 11.

This document is partially derived from earlier Mach documents written by Eric Cooper, Richard P. Draves and Randall Dean.

Contents

CHAPTER 1 **Introduction** 1
Interface Descriptions 1
Interface Types 2
Special Forms 3
Parameter Types 3

CHAPTER 2 **Library Support Functions** 5
MACH_PORT_VALID 6
environment_port 7
mach_device_server_port 8
mach_error 9
mach_error_string 10
mach_error_type 11
mach_init 12
mach_msg_destroy 13
mach_msg_server 14
mach_privileged_host_port 16
mach_task_self 17
mig_dealloc_reply_port 18
mig_get_reply_port 19
mig_init 20
mig_reply_setup 21
mig_strncpy 23
name_server_port 24
quit 25
round_page 26
service_port 27
slot_name 28
trunc_page 29
vm_page_size 30

CHAPTER 3 **C Thread Functions** 31
condition_alloc 32
condition_broadcast 33
condition_clear 34
condition_free 35
condition_init 36
condition_name 37

condition_set_name 38
condition_signal. 39
condition_wait 40
cthread_count. 42
cthread_data 43
cthread_detach 44
cthread_exit 45
cthread_fork 46
cthread_init 47
cthread_join 48
cthread_kernel_limit 49
cthread_limit 50
cthread_mach_msg 51
cthread_msg_active 53
cthread_msg_busy 54
cthread_name 55
cthread_self 56
cthread_set_data. 57
cthread_set_kernel_limit 58
cthread_set_limit 59
cthread_set_name. 60
cthread_stack_size 61
cthread_unwire. 62
cthread_wire. 63
cthread_yield 64
mutex_alloc 65
mutex_clear 66
mutex_free 67
mutex_init 68
mutex_lock. 69
mutex_name. 70
mutex_set_name 71
mutex_try_lock 72
mutex_unlock. 73
spin_lock 74
spin_try_lock 75
spin_unlock 76

CHAPTER 4 **Name Server** 77
netname_check_in 78
netname_check_out 80
netname_look_up. 82

netname_version 84

CHAPTER 5 **NetMemory Server**. 85
netmemory_cache 86
netmemory_create 88
netmemory_destroy 90

CHAPTER 6 **Service Server**. 91
service_checkin 92
service_waitfor. 93

APPENDIX A **C Language Functions**. 95
libmach_sa.a. 95
libthreads.a 96
libmach.a 96

APPENDIX B **Data Structure Definitions** 99
mig_reply_header 100

APPENDIX C **Error Return Values** 101

APPENDIX D **Index** 103

CHAPTER 1 Introduction

This book documents the interfaces of use in writing a Mach server. The text describes each interface in isolation. The relationship of interfaces to one another, and the way that interfaces are combined to write user servers is the subject of the *Server Writer's Guide*.

Interface Descriptions

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

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

  The Mach 3 system provide two general purpose libraries, and a few special purpose ones. The general purpose libraries are **libmach_sa.a** and **libmach.a**. **libmach_sa** (stand-alone) contains only kernel interfaces and a handful of library routines completely implemented without assistance from other servers. **libmach.a** contains all of the functions in **libmach_sa.a** plus other libraries that may call upon services of other servers (a BSD server, in particular). A server that wishes to “stand alone” in the absence of a BSD server or any of the Mach servers should link only against **libmach_-sa**. Many of the examples in the *Server Writer's Guide*, though, depend on other services (the ability to print to a terminal, for example) and therefore link against **libmach**.
- A synopsis of the interface, in C form
- An extended description of the function performed by the call

- Any macro or special forms of the call
- A description of each parameter to the call
- Additional notes on the use of the interface
- Cautions relating to the interface use
- An explanation of the significant return values
- References to related interfaces

Interface Types

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

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

Most of these interfaces are of the type **Function**. This means that there is actually a C callable function (most likely in **libmach.a**) that has the calling sequence listed and that when called invokes some function or sends a message to some server.

Some interfaces have the type **Server Interface**. Such a description applies to interfaces that are called in server tasks on behalf of messages sent from some other source. That is, it is assumed that some task is listening (probably with **mach_msg_server**) on a port to which a server is to send messages. A received message will be passed to a MIG generated server routine (*service*__**server**) which will call an appropriate server target function. It is these server target functions, one for each different message that can be received, that are listed as **Server Interface**s. For any given message, there are any number of possible server interface calling sequences that can be generated, by permuting the order of the data provided in the message, omitting some data elements or including or omitting various header field elements (such as sequence numbers). In most cases, a single server interface calling sequence has been chosen with a given MIG generated server message de-multiplexing routine that calls these interfaces. In some cases, there are more than one MIG generated server routines which call upon different server interfaces associated with that MIG service routine. In any event, all **Server Interfaces** contain within their documentation the name of the MIG generated server routine that invokes the interface.

Special Forms

There are various special interface forms defined in this volume.

- The **MACRO** form specifies macros (typically defined in **mach.h**) that provide shorthand equivalents for some variations of the longer function call.
- The **SEQUENCE NUMBER** form of a **Server Interface** defines an additional MIG generated interface that supplies the sequence number from the message causing the server interface to be invoked. The existence of such a form implies the existence of an alternate MIG generated message de-multiplexing routine that invokes this special interface form.
- The **ASYNCHRONOUS** form defines a MIG generated version of a **Function** that allows the function to be invoked asynchronously. Such a version requires an additional parameter to specify the reply port to which the reply is sent. The return value from the asynchronous function is the return status from the **mach_msg** call sending the request, not the resulting status of the target operation. The asynchronous interface also requires a matching **Server Interface** that defines the reply message containing data that would have been output values from the normal function, as well as the resulting status from the target operation.

Parameter Types

Each interface description supplies the C type of the various parameters. The parameter descriptions then indicate whether these parameters are input (“in”), output (“out”) or both (“in/out”). This information appears in square brackets before the parameter description. Additional information also appears within these brackets for special or non-obvious parameter conventions.

The most common notation is “scalar”, which means that the parameter somehow derives from an *int* type. Note that port types are of this form.

If the notation says “structure”, the parameter is a direct structure type whose layout is described in APPENDIX B unless the structure type is intended to be opaque.

The notation “pointer to in array/structure/scalar” means that the caller supplies a pointer to the data. Arrays always have this property following from C language rules. If not so noted, input parameters are passed by value.

Output parameters are always passed by reference following C language rules. Hence the notation “out array/structure/scalar” actually means that the caller must supply a pointer to the storage to receive the output value. If a parameter is in/out, the notation “pointer to in/out array/structure/scalar” will appear. Since the parameter is also an output parameter, it must be passed by reference, hence it appears as a “pointer to in array/structure/scalar” when used as an input parameter.

In contrast, the notation “out pointer to dynamic array” means that the target will allocate space for returned data (as if by **vm_allocate**) and will modify the pointer named by the output parameter (that is, the parameter to the function is a pointer to a pointer) to point to this allocated memory. The task should **vm_deallocate** this space when done referencing it.

For a Server Interface, the corresponding version of the above is “in pointer to dynamic array”. This indicates that the target has allocated space for the data (as if by **vm_allocate**) and is supplying a pointer to the data as the input parameter to the server interface routine. It is the job of the server interface routine to arrange for this data to be **vm_deallocate**d when the data is no longer needed.

An “unbounded out in-line array” specifies the variable in-line/out-of-line (referred to as unbounded in-line) array feature of MIG described in the *Server Writer’s Guide*. The caller supplies a pointer to a pointer whose value contains the address of an array whose size is specified in some other parameter (or known implicitly). Upon return, if this target pointer no longer points to the caller’s array (most likely because the caller’s array was not sufficiently large to hold the return data), then the target allocated space (as if by **vm_allocate**) into which the data was placed; otherwise, the data was placed into the supplied array.

CHAPTER 2 Library Support Functions

This chapter describes support functions and macros found in **libmach.a** and **<mach.h>**.

MACH_PORT_VALID

Macro — Determine if a port name names a valid port right

LIBRARY

#include <**mach.h**>

SYNOPSIS

```
boolean_t MACH_PORT_VALID
                (mach_port_t                                    right);
```

DESCRIPTION

The MACH_PORT_VALID macro determines if the specified port name names a valid port right.

PARAMETERS

right
: [in scalar] Port name

RETURN VALUE

FALSE if the specified name is MACH_PORT_NULL or MACH_-PORT_DEAD, TRUE otherwise

environment_port

Global Variable — Names the port to the environment server

LIBRARY

libmach.a

#include <**mach.h**>

SYNOPSIS

```
extern mach_port_t                                    environment_port;
```

DESCRIPTION

The **environment_port** variable contains the port name of a send right to the environment server. It is initialized by **mach_init** from the task's set of registered ports.

RELATED INFORMATION

Functions: **mach_ports_register**.

mach_device_server_port

Function — Finds the privileged kernel device master server port

LIBRARY

libmach.a

#include <**mach_privileged_ports.h**>

SYNOPSIS

```
mach_port_t mach_device_server_port
                    ();
```

DESCRIPTION

The **mach_device_server_port** function locates the privileged device master server port. This port allows the holder to open any device on the node. This function will succeed only for privileged tasks.

The call tries to find the device master port first by sending a special message (ID 999999) to the task's bootstrap port, and, failing that, through the undocumented CMU system call, **task_by_pid** (–33).

PARAMETERS

None

RETURN VALUE

Send rights to the device master server port or MACH_PORT_NULL

mach_error

Function — Print a Mach related error message

LIBRARY

libmach.a

#include <**mach_error.h**>

SYNOPSIS

```
void mach_error
                (char*                                  string,
                kern_return_t                            errno);
```

DESCRIPTION

The **mach_error** function prints a Mach related error message on *standard error.* The message consists of *string* followed by **mach_error_string** (*errno*) followed by *errno*. The actual error code is included in case it is bogus.

PARAMETERS

string

[pointer to in array of *char*] A string to prefix to the error message

errno

[in scalar] A return code from a Mach invocation

RETURN VALUE

None

RELATED INFORMATION

error(5), **mach_error_string**, **mach_error_type**.

mach_error_string

Function — Return a human readable error string

LIBRARY

libmach.a

#include <**mach_error.h**>

SYNOPSIS

char* **mach_error_string**
(kern_return_t *errno*);

DESCRIPTION

The **mach_error_string** function returns a human readable string corresponding to the specified Mach return value. This string is statically allocated in the Mach library.

PARAMETERS

errno
[in scalar] A return code from a Mach invocation

RETURN VALUE

A pointer to the error message string

RELATED INFORMATION

error(5), **mach_error**, **mach_error_type**.

mach_error_type

Function — Return the system and subsystem name for an error

LIBRARY

libmach.a

#include <**mach_error.h**>

SYNOPSIS

```
char* mach_error_type
                (kern_return_t                                   errno);
```

DESCRIPTION

The **mach_error_string** function returns a string containing the system and subsystem name that produced the specified Mach return value. This string is statically allocated in the Mach library.

PARAMETERS

errno

[in scalar] A return code from a Mach invocation

RETURN VALUE

A pointer to the system name string

RELATED INFORMATION

error(5), **mach_error**, **mach_error_string**.

mach_init

Function — Mach task related start-up.

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
int mach_init
                ();
```

DESCRIPTION

The **mach_init** function performs MACH related task start-up. It also invokes MIG related start-up. This call is made by **_start** automatically when a task starts.

PARAMETERS

None

RETURN VALUE

Not meaningful.

RELATED INFORMATION

Functions: **_start**, **mig_init**.

mach_msg_destroy

Function — Clean up data associated with a received message

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
void mach_msg_destroy
                (mach_msg_header_t*                              msg);
```

DESCRIPTION

The **mach_msg_destroy** function de-allocates all port rights and out-of-line memory found in a received message. Send and send-once rights are de-allocated; receive rights have their reference count decremented.

PARAMETERS

msg
: [pointer to in structure] A received message.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **mach_msg**.

Data Structures: **mach_msg_header**.

mach_msg_server

Function — A simple generic server message loop

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
mach_msg_return_t mach_msg_server
                (boolean_t                                    (*demux)
                                         (mach_msg_header_t* request,
                                           mach_msg_header_t* reply),
                mach_msg_size_t                              max_size,
                mach_port_t                                  rcv_name);
```

DESCRIPTION

The **mach_msg_server** function loops, reading messages from *rcv_name*, and passing them to the *demux* routine. The *demux* routine is called as follows:

(**demux*) (*request*, *reply*);

where:

request
: [pointer to in structure] is a pointer to the message received from *rcv_name*.

reply
: [out structure] is a pointer to an area (of size *max_size*) into which a reply message is to be placed.

The *demux* routine is declared to take **mach_msg_header_t** as arguments. It is actually passed **mig_reply_header_t** values which are cast accordingly.

A reply message will be sent only if the value for the *RetCode* field of the *reply* structure is KERN_SUCCESS and the value of the *msgh_remote_port* in the *reply* structure is other than MACH_PORT_NULL. An error in the message send or receive operation of other than MACH_SEND_INVALID_DEST terminates the loop. The resultant error code is returned.

PARAMETERS

demux
[in scalar] A pointer to a routine to be called for each message received.

max_size
[in scalar] The maximum size message to receive.

rcv_name
[in scalar] A receive right to a port.

RETURN VALUE

KERN_RESOURCE_SHORTAGE
Insufficient virtual address space for the receive and reply buffers.

Other MIG and **mach_msg** errors terminate the call.

RELATED INFORMATION

Functions: **mach_msg**, **mig_reply_setup**.

Data structures: **mach_msg_header**, **mig_reply_header**.

mach_privileged_host_port

Function — Finds the privileged host control port

LIBRARY

libmach.a

#include <**mach_privileged_ports.h**>

SYNOPSIS

mach_port_t **mach_privileged_host_port**
();

DESCRIPTION

The **mach_privileged_host_port** function locates the privileged host control port. This port allows the holder to obtain rights to any other port on the node (with the exception of the device master port). This function will succeed only for privileged tasks.

The call tries to find the host control port first by sending a special message (ID 999999) to the task's bootstrap port, and, failing that, through the undocumented CMU system call, **task_by_pid** (–33).

PARAMETERS

None

RETURN VALUE

Send rights to the host control port or MACH_PORT_NULL

mach_task_self

Macro — Returns the task self port

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach.h**>

SYNOPSIS

```
mach_port_t mach_task_self
                 ();
```

DESCRIPTION

The **mach_task_self** macro returns send rights to the task's own port. The include file **<mach.h>** redefines the kernel function to simply return the value **mach_task_self_**, cached by the Mach run-time.

PARAMETERS

None

RETURN VALUE

Send rights to the task's port.

RELATED INFORMATION

Functions: **mach_task_self** (kernel call).

mig_dealloc_reply_port

Function — De-allocate the reply port for MIG interfaces

LIBRARY

libmach_sa.a, **libmach.a**, **libthreads.a**

Not declared anywhere.

SYNOPSIS

```
void mig_dealloc_reply_port
                    ();
```

DESCRIPTION

The **mig_dealloc_reply_port** function is called by MIG interfaces after a timeout on the reply port.

PARAMETERS

None

RETURN VALUE

None

RELATED INFORMATION

Functions: **mig_get_reply_port**.

mig_get_reply_port

Function — Generate a reply port for MIG interfaces

LIBRARY

libmach_sa.a, **libmach.a**, **libthreads.a**

Not declared anywhere.

SYNOPSIS

```
mach_port_t mig_get_reply_port
                 ();
```

DESCRIPTION

The **mig_get_reply_port** function is called by MIG interfaces when they need a reply port.

PARAMETERS

None

RETURN VALUE

The MIG reply port

RELATED INFORMATION

Functions: **mig_dealloc_reply_port**.

mig_init

Function — Prepares the task to perform MIG related MACH IPC functions

LIBRARY

libmach_sa.a, **libmach.a**, **libthreads.a**

Not declared anywhere.

SYNOPSIS

```
void mig_init
                ();
```

DESCRIPTION

The **mig_init** function prepares the task to use MIG related services. This call is made automatically via **_start** when the task begins.

PARAMETERS

None

RETURN VALUE

None.

RELATED INFORMATION

Functions: **_start**, **cthread_init**.

mig_reply_setup

Function — Initialize a MIG reply message

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
void mig_reply_setup
                (mach_msg_header_t*                               request,
                 mach_msg_header_t*                                 reply);
```

DESCRIPTION

The **mig_reply_setup** function initializes the header of a reply message based upon the contents of a client's request for service. This initialization is normally done as part of the processing done by a MIG generated server de-multiplexing routine (normally *sys*_**server**). If, however, the MIG generated message typing routines (normally *sys*_**server_routine**) are used instead, **mig_reply_setup** would be used to perform the reply message initialization not done by these typing routines. Typical use is:

```
[1]  mig_reply_setup (&request, &reply);
[2]  if ((routine = sys1_server_routine (&request) != 0) ||
[3]                 (routine = sys2_server_routine (&request) != 0) ||
[4]                 (routine = sys3_server_routine (&request) != 0))
[5]       (*routine) (&request, &reply);
```

PARAMETERS

request
: [pointer to in structure] Request message from the client.

reply
: [out structure] Initialized reply message.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **mach_msg**, **mach_msg_server**.

Data structures: **mach_msg_header**, **mig_reply_header**.

mig_strncpy

Function — Copy a character string, null terminated, with count.

LIBRARY

libmach_sa.a, **libmach.a**

Not declared anywhere.

SYNOPSIS

```
int mig_strncpy
                (char*                                          dst,
                 char*                                          src,
                 unsigned int                                length);
```

DESCRIPTION

The **mig_strncpy** function copies a character string from *src* to *dst*. The copy terminates either when *length*-1 characters have been copied, or when a null character is encountered, whichever comes first. This routine differs from **strncpy** in that the resulting string is always null terminated.

PARAMETERS

dst
: [out array of *char*] Destination of the copy.

src
: [pointer to in array of *char*] Source of the copy.

length
: [in scalar] Number of bytes to move.

RETURN VALUE

Length of the resultant string, including the null terminating byte.

RELATED INFORMATION

Functions: **strncpy**.

name_server_port

Global Variable — Names the port to the name server

LIBRARY

libmach.a

#include <**mach.h**>

SYNOPSIS

```
extern mach_port_t                                    name_server_port;
```

DESCRIPTION

The **name_server_port** variable contains the port name of a send right to the name server. It is initialized by **mach_init** from the task's set of registered ports.

RELATED INFORMATION

Functions: **mach_ports_register**.

quit

Function — Print message and exit

LIBRARY

libmach.a

Not declared anywhere.

SYNOPSIS

```
void quit
               (int                                                    status,
               char*                                              format, ...);
```

DESCRIPTION

The **quit** function prints on *standard error* the message specified by the **printf** argument list *format*,... and then exits.

PARAMETERS

status

[in scalar] The process' return code.

format

[pointer to in array of *char*] A **printf** control string.

RETURN VALUE

None

RELATED INFORMATION

printf(3), **exit**(2), **wait**(2).

round_page

Macro — Round a virtual address to a page boundary

LIBRARY

#include <**mach.h**>

SYNOPSIS

```
vm_offset_t round_page
                (vm_offset_t                                   x);
```

DESCRIPTION

The **round_page** macro rounds its virtual address argument to the nearest page boundary.

PARAMETERS

x

[in scalar] Virtual address

RETURN VALUE

Rounded virtual address

RELATED INFORMATION

Functions: **trunc_page**.

service_port

Global Variable — Names the port to the service server

LIBRARY

libmach.a

#include <**mach.h**>

SYNOPSIS

```
extern mach_port_t                                       service_port;
```

DESCRIPTION

The **service_port** variable contains the port name of a send right to the service server. It is initialized by **mach_init** from the task's set of registered ports.

RELATED INFORMATION

Functions: **mach_ports_register**.

slot_name

Function — Converts CPU type and subtype to human readable form

LIBRARY

libmach.a

Not declared anywhere.

SYNOPSIS

```
void slot_name
        (cpu_type_t                     cpu_type,
         cpu_subtype_t                  cpu_subtype,
         char**                         cpu_name,
         char**                         cpu_subname);
```

DESCRIPTION

The **slot_name** function converts the specified *cpu_type* / *cpu_subtype* pair to their human readable counterparts. Two strings, which are statically allocated in the library, corresponding to the type and subtype are passed back to the caller in the *cpu_name* and *cpu_subname* parameters.

PARAMETERS

cpu_type

[in scalar] Type of the CPU, CPU_TYPE_VAX, CPU_TYPE_I386, etc.

cpu_subtype

[in scalar] Subtype of the CPU, CPU_SUBTYPE_VAX780, CPU_SUBTYPE_AT386, etc.

cpu_name

[out array of *char*] Corresponding CPU type name

cpu_subname

[out array of *char*] Corresponding CPU subtype name

RETURN VALUE

None

trunc_page

Macro — Truncate a virtual address to a page boundary

LIBRARY

#include <**mach.h**>

SYNOPSIS

```
vm_offset_t trunc_page
                (vm_offset_t                                        x);
```

DESCRIPTION

The **trunc_page** macro truncates its virtual address argument down to the nearest page boundary.

PARAMETERS

x
[in scalar] Virtual address

RETURN VALUE

Truncated virtual address

RELATED INFORMATION

Functions: **round_page**.

vm_page_size

Global Variable — Contains the page size for the current task.

LIBRARY

libmach_sa.a, **libmach.a**

#include <**mach.h**>

SYNOPSIS

```
extern vm_size_t                                        vm_page_size;
```

DESCRIPTION

The **vm_page_size** variable contains the task’s page size, in bytes.

RELATED INFORMATION

Functions: **vm_statistics**.

CHAPTER 3 C Thread Functions

This chapter describes functions that provide thread support for C programs.

Note that including **libthreads.a** redefines some system internal routines (**mig_init**, **mig_get_reply_port** and **mig_dealloc_reply_port**). **libthreads.a** must be linked prior to **libmach.a** when used.

All of the functions defined in this chapter are in **libthreads.a** and defined in <**cthreads.h**>.

condition_alloc

Macro — Dynamically allocate a condition variable

SYNOPSIS

```
condition_t condition_alloc
                    ();
```

DESCRIPTION

The **condition_alloc** macro dynamically allocates and initializes a condition variable.

PARAMETERS

None

RETURN VALUE

A pointer to the condition variable.

RELATED INFORMATION

Functions: **condition_free**.

condition_broadcast

Macro — Broadcast a status change in a condition variable

SYNOPSIS

```
void condition_broadcast
        (condition_t                      c);
```

DESCRIPTION

The **condition_broadcast** macro indicates that a status change has occurred associated with condition variable *c*. All C threads waiting for this condition variable will be wakened.

PARAMETERS

c
: [pointer to in structure] A condition variable indicating the status change

NOTES

The mutex named in the corresponding **condition_wait** call must be held during this call or the results are unspecified.

RETURN VALUE

None

RELATED INFORMATION

Functions: **condition_signal**, **condition_wait**.

condition_clear

Macro — Finalizes use of a user allocated condition variable

SYNOPSIS

void **condition_clear**
        (condition_t   *c*);

DESCRIPTION

The **condition_clear** macro finalizes use of a user allocated condition variable. In this context, a user allocated variable is one not obtained via **condition_alloc** (one initialized with **condition_init**). Finalizing a condition variable is also considered to broadcast the condition so associated.

PARAMETERS

c
    [pointer to in structure] A condition variable

RETURN VALUE

None

RELATED INFORMATION

Functions: **condition_init**.

condition_free

Macro — Free a dynamically allocated condition variable

SYNOPSIS

```
void condition_free
                (condition_t                                          c);
```

DESCRIPTION

The **condition_free** macro frees a dynamically allocated condition variable (one obtained with **condition_alloc**). Freeing a condition variable is considered to broadcast the condition so associated.

PARAMETERS

c

[pointer to in structure] A condition variable

RETURN VALUE

None

RELATED INFORMATION

Functions: **condition_alloc**.

condition_init

Macro — Initialize a user allocated condition variable

SYNOPSIS

```
void condition_init
        (condition_t                              c);
```

DESCRIPTION

The **condition_init** macro initializes a user allocated condition variable. In this context, a user allocated variable is one not obtained via **condition_alloc**.

PARAMETERS

c

[pointer to in structure] A condition variable

RETURN VALUE

None

RELATED INFORMATION

Functions: **condition_clear**.

condition_name

Macro — Return a name associated with a condition variable

SYNOPSIS

```
char* condition_name
                (condition_t                                    c);
```

DESCRIPTION

The **condition_name** macro returns the name associated with the given condition variable.IF there is no associated name, “?” is returned.

PARAMETERS

c

[pointer to in structure] A condition variable

RETURN VALUE

A pointer to the associated name

RELATED INFORMATION

Functions: **condition_set_name**.

condition_set_name

Macro — Associate a name with a condition variable

SYNOPSIS

```
void condition_set_name
                   (condition_t                                   c,
                   char*                                       name);
```

DESCRIPTION

The **condition_set_name** macro associates a name with a condition variable. Currently, these names are not used for anything; they can be retrieved with **condition_name**. Note that only a pointer to the name is associated with the condition variable; the name string must not be de-allocated until the name association is broken.

PARAMETERS

c

[pointer to in structure] A condition variable

name

[pointer to in array of *char*] Name to associate

RETURN VALUE

None

RELATED INFORMATION

Functions: **condition_name**.

condition_signal

Macro — Signal that a condition has occurred

SYNOPSIS

```
void condition_signal
                (condition_t                                        c);
```

DESCRIPTION

The **condition_signal** macro indicates that a status change has occurred associated with condition variable *c*. At least one C thread waiting for this condition variable will be wakened.

PARAMETERS

c

[pointer to in structure] A condition variable indicating the status change

NOTES

The mutex named in the corresponding **condition_wait** call must be held during this call or the results are unspecified.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **condition_broadcast**, **condition_wait**.

condition_wait

Function — Wait for a status change associated with a condition variable

SYNOPSIS

```
void condition_wait
            (condition_t                           c,
             mutex_t                               m);
```

DESCRIPTION

The **condition_wait** function waits for a status change associated with some shared data. The calling thread is assumed to hold a mutex, *m*, protecting the data locked. This call releases the mutex and waits for the condition variable *c* to be signaled, indicating a change to the shared data. This call returns at some point in time after this event, with the mutex once again locked. The time between the signalling of the condition variable and the locking of the mutex is arbitrary; it is possible that some other thread could have locked the mutex and performed yet other changes (and condition signalling) prior to this thread re-obtaining the mutex.

PARAMETERS

c

[pointer to in structure] The condition variable indicating the status change

m

[pointer to in structure] A mutex that locks the data associated with the condition variable

NOTES

The typical use of this function is in a loop as follows:

```
mutex_t m;
condition_t c;
mutex_lock (m);
while (...status of shared data is not okay...)
        condition_wait (c, m);
...use shared data...
mutex_unlock (m);
```

RETURN VALUE

None.

RELATED INFORMATION

Functions: **condition_signal**, **condition_broadcast**.

cthread_count

Function — Return the current number of C thread

SYNOPSIS

int **cthread_count**
();

DESCRIPTION

The **cthread_count** function returns the current number of C threads. A C thread is counted as no longer existing when it returns from its top-level function (calls **cthread_exit**), not when some other thread detaches it or joins with it.

PARAMETERS

None

RETURN VALUE

Number of current C threads

RELATED INFORMATION

Functions: **cthread_fork**, **cthread_exit**.

cthread_data

Macro — Returned data associated with a thread

SYNOPSIS

```
any_t cthread_data
                (cthread_t                                              t);
```

DESCRIPTION

The **cthread_data** macro returns the data value associated with the given thread. This value provides a simple form of thread-specific “global” data. More elaborate mechanisms may be built upon this single value.

After a thread exits, any attempt to get or set its associated data is illegal, so any de-allocation or other cleanup of the data must be done before the thread exits. It is always safe to access the data associated with the caller’s own thread (**cthread_self**), or with a thread that has not yet been joined or detached.

PARAMETERS

t

[pointer to in structure] A thread identifier

RETURN VALUE

The thread’s associated data value

RELATED INFORMATION

Functions: **cthread_set_data**.

cthread_detach

Function — Detach a C thread from all threads

SYNOPSIS

```
void cthread_detach
        (cthread_t                    t);
```

DESCRIPTION

The **cthread_detach** function indicates that thread **t** will never be joined.

PARAMETERS

t

[pointer to in structure] Thread identifier

NOTES

Since the fact that a thread is to be detached is normally known when it is created, this call is normally used as: **cthread_detach**(**cthread_fork**(*func*, *arg*));

RETURN VALUE

None.

RELATED INFORMATION

Functions: **cthread_fork**, **cthread_join**.

cthread_exit

Function — Terminate the current C thread

SYNOPSIS

```
void cthread_exit
        (any_t                                    result);
```

DESCRIPTION

The **cthread_exit** function terminates the calling thread. This call is implicit when the top-level function of a thread returns, in which case the argument to **cthread_exit** is the return value from the top-level function, but it can also be called explicitly. The *result* is made available to a thread that joins with this thread (**cthread_join**), or discarded if the thread is detached. If this is the first (main) thread, its termination will not terminate the task, but will instead wait for all other C threads to terminate and then terminate the task. The exit status for the task becomes the value of *result*.

PARAMETERS

result

[in scalar] A value to be given to **cthread_join**

RETURN VALUE

None.

RELATED INFORMATION

Functions: **cthread_fork**, **cthread_join**, **cthread_detach**.

cthread_fork

Function — Create a new C thread

SYNOPSIS

cthread_t **cthread_fork**
(any_t (**func*) (any_t *arg*), any_t *arg);*

DESCRIPTION

The **cthread_fork** function creates a new thread which will execute concurrently with the current thread. This is the sole means of creating new threads. The new thread will execute the following call:

result = (**func*) (*arg*);

The result value from this call (assuming the call does not terminate itself via **cthread_exit**) is available via **cthread_join**. The call to **cthread_fork** returns a thread identifier useful for a call to **cthread_detach** or **cthread_join**. A thread may either be joined or detached only once. If the thread is neither joined nor detached, the thread's associated data will never be released.

PARAMETERS

func
[in scalar] Top-level function to execute in the new thread.

arg
[in scalar] Single argument to pass to *func*.

RETURN VALUE

A thread identifier naming the new thread.

RELATED INFORMATION

Functions: **cthread_exit**, **cthread_join**, **cthread_detach**.

cthread_init

Function — Initialize the C threads package.

SYNOPSIS

int **cthread_init**
();

DESCRIPTION

The **cthread_init** function initializes the C threads package. It is automatically called by **_start** when the C threads package is included when linking. This call also initializes the multi-threaded MIG routines. After this call, the initial point of control in the task becomes the first C thread. When this first thread terminates, it does not immediately terminate the task. It waits for all threads to terminate before so doing. The exit status for the task becomes the thread exit status of this first (main) thread.

PARAMETERS

None

RETURN VALUE

The top of the first thread's stack. (The return type of this function is incorrect.)

RELATED INFORMATION

Functions: **_start**.

cthread_join

Function — Join with a C thread

SYNOPSIS

```
any_t cthread_join
        (cthread_t                                t);
```

DESCRIPTION

The **cthread_join** function suspends the caller until the thread *t* terminates.

PARAMETERS

t

[pointer to in structure] A thread identifier

RETURN VALUE

Either the result of *t*'s top-level function (if it returned normally) or the argument with which *t* explicitly called **cthread_exit**

RELATED INFORMATION

Functions: **cthread_fork**, **cthread_detach**, **cthread_exit**.

cthread_kernel_limit

Function — Get the kernel thread limit for C threads

LIBRARY

Not defined anywhere.

SYNOPSIS

int **cthread_kernel_limit**
();

DESCRIPTION

The **cthread_kernel_limit** function returns the current limit on the number of kernel threads to use to support C threads. A value of zero is considered as no limit.

PARAMETERS

None

RETURN VALUE

The current kernel thread limit.

RELATED INFORMATION

Functions: **cthread_set_kernel_limit**, **cthread_limit**, **cthread_set_limit**.

cthread_limit

Function — Return the limit on active C threads

SYNOPSIS

```
int cthread_limit
                ();
```

DESCRIPTION

The **cthread_limit** function returns the limit on the number of active C threads. In this context, a C threads is considered as active if it can be considered for execution by a supporting kernel thread (that is, it has an assigned *cproc*). The actual number of C threads that can actually be in execution at any time is governed by the thread kernel limit. A value of zero is considered as no limit.

PARAMETERS

None

RETURN VALUE

The number of allowed active C threads

RELATED INFORMATION

Functions: **cthread_set_limit**, **cthread_kernel_limit**, **cthread_set_kernel_limit**.

cthread_mach_msg

Function — C thread optimized MACH message routine

LIBRARY

Not defined anywhere.

SYNOPSIS

```
mach_msg_return_t cthread_mach_msg
                (mach_msg_header_t                          msg,
                 mach_msg_option_t                       option,
                 mach_msg_size_t                      send_size,
                 mach_msg_size_t                       rcv_size,
                 mach_port_t                           rcv_name,
                 mach_msg_timeout_t                     timeout,
                 mach_port_t                             notify,
                 int                                        min,
                 int                                       max);
```

DESCRIPTION

The **cthread_mach_msg** function performs a **mach_msg** call. (It is assumed that the MACH_RCV_MSG is specified.) This call differs from **mach_msg** only in as much as that this call limits the number of threads that may be actively servicing a given port (or port set). In this sense, “actively servicing” means that the C thread is allowed to wait for a message from the port. This call, as well as **cthread_msg_active**, declare a C thread to be actively servicing a port. When a C thread blocks performing a kernel function (such as **mach_msg**), it blocks its underlying MACH thread as well. If this thread’s waiting would exceed the *max* value established at the first wait from the port, this thread will send its message (if there is one), but the C thread will block instead of doing the receive at this time. In this way, the underlying MACH thread is free to perform other work. When the number of C threads (and corresponding MACH threads) actively servicing this port falls below the *min* value, a C thread will be wakened to then perform its message receive (and become an active listener for this port as a result), blocking itself in the message receive (and thereby blocking its MACH thread as well).

A C thread cease to be an active receiver for a port when it calls **cthread_mach_msg** with some different port, or when it calls **cthread_msg_busy**.

PARAMETERS

msg

[pointer to in/out structure] A message buffer. This should be aligned on a long-word boundary.

option

[in scalar] Refer to **mach_msg** for a description of this parameter.

send_size

[in scalar] Refer to **mach_msg** for a description of this parameter.

rcv_size

[in scalar] Refer to **mach_msg** for a description of this parameter.

rcv_name

[in scalar] Refer to **mach_msg** for a description of this parameter.

timeout

[in scalar] Refer to **mach_msg** for a description of this parameter.

notify

[in scalar] Refer to **mach_msg** for a description of this parameter.

min

[in scalar] The maximum number of threads that can be left waiting for messages from *rcv_name* before other threads are released.

max

[in scalar] The maximum number of threads that can be waiting for messages from *rcv_name*.

RETURN VALUE

Return value from the **mach_msg** call.

RELATED INFORMATION

Functions: **mach_msg**, **cthread_msg_active**, **cthread_msg_busy**.

cthread_msg_active

Function — Mark this thread as actively servicing a port

LIBRARY

Not defined anywhere.

SYNOPSIS

```
void cthread_msg_active
        (mach_port_t                    port,
        int                             min,
        int                             max);
```

DESCRIPTION

The **cthread_msg_active** function declares that this C thread will be actively receiving (and thereby waiting for) messages from the specified port. By performing this call prior to any **cthread_mach_msg** calls for *port*, this thread is reserved as a listener for the port, and is guaranteed that it can receive without its C thread being blocked waiting for other threads to cease being active receivers for this port.

PARAMETERS

port

[in scalar] Receive port this thread will service

min

[in scalar] The maximum number of threads that can be left waiting for messages from *port* before other threads are released.

max

[in scalar] The maximum number of threads that can be waiting for messages from *port*.

RETURN VALUE

None

RELATED INFORMATION

Functions: **mach_msg**, **cthread_mach_msg**, **cthread_msg_busy**.

cthread_msg_busy

Function — Cease to be an active receiver for a port

LIBRARY

Not defined anywhere.

SYNOPSIS

```
void cthread_msg_busy
            (mach_port_t                         port,
             int                                 min,
             int                                 max);
```

DESCRIPTION

The **cthread_msg_busy** function declares that this C thread will no longer be an active listener for its port. A thread is declared an active listener for a port either via **cthread_mach_msg** or **cthread_msg_active**. If, by releasing active listenership (either via this call or a **cthread_mach_msg** call specifying a different port), the active listeners falls below the minimum value for the port, a C thread will be wakened so it can perform its receive operation.

PARAMETERS

port

[in scalar] Port for which this thread will no longer be a receiver

min

[in scalar] Duplicates to match **cthread_msg_active**.

max

[in scalar] Duplicates to match **cthread_msg_active**.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **mach_msg**, **cthread_msg_active**, **cthread_mach_msg**.

cthread_name

Function — Return the name associated with a thread

SYNOPSIS

```
char* cthread_name
               (cthread_t                                    t);
```

DESCRIPTION

The **cthread_name** function returns a pointer to the name associated with a thread. If the thread has no associated name, “?” is returned.

PARAMETERS

t

[pointer to in structure] A thread identifier

RETURN VALUE

The thread’s associated name

RELATED INFORMATION

Functions: **cthread_set_name**.

cthread_self

Macro — Return the caller’s thread identifier

SYNOPSIS

cthread_t **cthread_self**
();

DESCRIPTION

The **cthread_self** macro returns the caller’s own thread identifier, which is the same value that was returned by **cthread_fork** to the creator of the thread. The thread identifier uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be re-used by the underlying implementation, the programmer should be careful to clean up such associations when threads exit

PARAMETERS

None

RETURN VALUE

The thread’s own identifier

RELATED INFORMATION

Functions: **cthread_fork**.

cthread_set_data

Macro — Associate a data value with a thread

SYNOPSIS

```
void cthread_set_data
        (cthread_t                                   t,
         any_t                                    data);
```

DESCRIPTION

The **cthread_set_data** macro associates a single data value with a thread. This value may be subsequently retrieved by **cthread_data**.

PARAMETERS

t

[pointer to in structure] A thread identifier

data

[in scalar] A single data value to associate with the thread

RETURN VALUE

None

RELATED INFORMATION

Functions: **cthread_data**.

cthread_set_kernel_limit

Function — Set the maximum number of kernel threads for C threads

LIBRARY

Not defined anywhere.

SYNOPSIS

void **cthread_set_kernel_limit**
(int *n*);

DESCRIPTION

The **cthread_set_kernel_limit** function sets the limit on the number of kernel threads to use to support C threads. If the current number of kernel threads exceeds this value, none are destroyed. A value of zero is considered as no limit.

PARAMETERS

n

[in scalar] Maximum number of kernel threads

RETURN VALUE

None.

RELATED INFORMATION

Functions: **cthread_kernel_limit**, **cthread_limit**, **cthread_set_limit**.

cthread_set_limit

Function — Set the limit of active C threads

SYNOPSIS

```
void cthread_set_limit
        (int                                   n);
```

DESCRIPTION

The **cthread_set_limit** function limits the number of active C threads. In this context, a C thread is considered as active if it can be considered for execution by a supporting kernel thread (that is, it has an assigned *cproc*). The actual number of C threads that can actually be in execution at any time is governed by the thread kernel limit. A value of zero is considered as no limit.

PARAMETERS

n
[in scalar] Limit on active C threads

RETURN VALUE

None

RELATED INFORMATION

Functions: **cthread_limit**, **cthread_kernel_limit**, **cthread_set_kernel_limit**.

cthread_set_name

Function — Associate a name with a thread

SYNOPSIS

```
void cthread_set_name
        (cthread_t                          t,
         char*                              name);
```

DESCRIPTION

The **cthread_set_name** function associates a name with a thread. Currently, these names are not used for anything; they can be retrieved with **cthread_name**. The initial thread is automatically given a name of “main”. Note that only a pointer to the name is associated with the thread; the name string must not be de-allocated until the name association is broken.

PARAMETERS

t

[pointer to in structure] A thread identifier

name

[pointer to in array of *char*] A name to associate

RETURN VALUE

None

RELATED INFORMATION

Functions: **cthread_name**.

cthread_stack_size

Global Variable — Size (in bytes) of the stack allocated to a C thread

SYNOPSIS

```
extern vm_size_t                                          cthread_stack_size;
```

DESCRIPTION

The **cthread_stack_size** variable contains the size in bytes of a C thread's stack. This value is normally initialized to a default value when the task is initialized. It may be set to a value at compile time by declaring:

vm_size_t **cthread_stack_size** = *N*;

NOTES

cthread_stack_size must be a multiple of the system page size.

cthread_unwire

Function — Un-bind a C thread from a MACH thread.

LIBRARY

Not defined anywhere.

SYNOPSIS

```
void cthread_unwire
                ();
```

DESCRIPTION

The **cthread_unwire** function breaks the binding of the current C thread from its MACH thread. After this call, the MACH thread is free to service any unbound C thread, and this C thread may be serviced by any unbound MACH thread.

PARAMETERS

None

RETURN VALUE

None.

RELATED INFORMATION

Functions: **cthread_wire**.

cthread_wire

Function — Bind a C thread to a MACH thread

LIBRARY

Not defined anywhere.

SYNOPSIS

void **cthread_wire**
();

DESCRIPTION

The **cthread_wire** function binds the calling C thread to the MACH thread currently executing it. After this, the current MACH thread is dedicated to running only this C thread, and this C thread will run using only this MACH thread. This is done to guarantee a free MACH thread for the activities of this C thread.

PARAMETERS

None

RETURN VALUE

None.

RELATED INFORMATION

Functions: **cthread_unwire**.

cthread_yield

Function — Schedule another thread

SYNOPSIS

```
void cthread_yield
                  ();
```

DESCRIPTION

The **cthread_yield** function provides a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor. If the current C thread is bound to a MACH thread, this call is equivalent to **switch_pri**. Otherwise, an attempt is made to use this MACH thread to service some other C thread; if no such runable C thread exists, **switch_pri** is called. Since multiple C threads can be serviced by a single MACH thread, and there is no pre-emption mechanism that will forcibly provide this servicing, this call may be needed to avoid starvation of threads.

PARAMETERS

None

RETURN VALUE

None.

RELATED INFORMATION

Functions: **swtch**, **swtch_pri**, **thread_switch**.

mutex_alloc

Macro — Allocate a mutex variable

SYNOPSIS

mutex_t **mutex_alloc**
();

DESCRIPTION

The **mutex_alloc** macro allocates heap storage properly constructed as a mutex variable.

PARAMETERS

None

RETURN VALUE

A pointer to a mutex

RELATED INFORMATION

Functions: **mutex_free**.

mutex_clear

Macro — Finalize use of a user allocated mutex variable

SYNOPSIS

```
void mutex_clear
        (mutex_t                                        m);
```

DESCRIPTION

The **mutex_clear** macro finalizes the use of a user allocated mutex variable. A user allocated mutex here means one for which the storage was obtained by the user in ways other than **mutex_alloc**, and subsequently initialized by **mutex_init**.

PARAMETERS

m
[pointer to in structure] A mutex

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_init**.

mutex_free

Macro — Free a dynamically allocated mutex variable

SYNOPSIS

```
void mutex_free
                (mutex_t                                    m);
```

DESCRIPTION

The **mutex_free** macro frees a dynamically allocated mutex variable obtained via **mutex_alloc**.

PARAMETERS

m

[pointer to in structure] A mutex

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_alloc**.

mutex_init

Macro — Initialize a user allocated mutex variable

SYNOPSIS

```
void mutex_init
        (mutex_t                                    m);
```

DESCRIPTION

The **mutex_init** macro initializes user allocated storage to be a mutex variable. In this context, user allocated storage is meant to be any storage other than that obtained via **mutex_alloc**.

PARAMETERS

m

[pointer to in structure] A mutex variable

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_clear**.

mutex_lock

Macro — Lock a mutex

SYNOPSIS

```
void mutex_lock
                (mutex_t                                        m);
```

DESCRIPTION

The **mutex_lock** macro locks the specified mutex. It blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until *m* is unlocked. The case of a thread attempting to lock a mutex it already holds is *not* treated specially; deadlock will result.

PARAMETERS

m

[pointer to in structure] A mutex

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_try_lock**, **mutex_unlock**.

mutex_name

Macro — Return the name associated with a mutex

SYNOPSIS

char* **mutex_name**
(mutex_t *m*);

DESCRIPTION

The **mutex_name** macro returns the name associated with a mutex. It the mutex has no associated name, “?” is returned.

PARAMETERS

m
[pointer to in structure] A mutex

RETURN VALUE

The mutex’ associated name

RELATED INFORMATION

Functions: **mutex_set_name**.

mutex_set_name

Macro — Associate a name with a mutex

SYNOPSIS

```
void mutex_set_name
                (mutex_t                                        m,
                char*                                        name);
```

DESCRIPTION

The **mutex_set_name** macro associates a name with a mutex variable. Currently, these names are not used for anything; they can be retrieved with **mutex_name**. Note that only a pointer to the name is associated with the mutex variable; the name string must not be de-allocated until the name association is broken.

PARAMETERS

m

[pointer to in structure] A mutex

name

[pointer to in array of *char*] Name to associate with *m*

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_name**.

mutex_try_lock

Macro — Attempt to lock a mutex

SYNOPSIS

```
boolean_t mutex_try_lock
          (mutex_t                                          m);
```

DESCRIPTION

The **mutex_try_lock** macro attempts to lock the mutex *m*, like **mutex_lock**. This macro does not block waiting for the mutex to become locked, returning a status in this case.

PARAMETERS

m
: [pointer to in structure] A mutex

RETURN VALUE

TRUE
: The mutex is locked to this thread.

FALSE
: The mutex is locked to some other thread.

RELATED INFORMATION

Functions: **mutex_lock**, **mutex_unlock**.

mutex_unlock

Macro — Unlock a mutex

SYNOPSIS

```
void mutex_unlock
                (mutex_t                                        m);
```

DESCRIPTION

The **mutex_unlock** macro unlocks the specified mutex, giving other threads a chance to lock it.

PARAMETERS

m

[pointer to in structure] A mutex

RETURN VALUE

None

RELATED INFORMATION

Functions: **mutex_lock**, **mutex_try_lock**.

spin_lock

Macro — Lock a spin lock.

SYNOPSIS

```
void spin_lock
        (spin_lock_t*                                    p);
```

DESCRIPTION

The **spin_lock** macro locks the specified spin lock. It does not return until the lock is locked to this thread. A spin lock is a lower overhead lock than a mutex, and as a result lacks some of the functionality of a mutex. A spin lock is so named because a thread waiting for the lock “spins”, wasting CPU time until the lock is released by the holding thread. (If the C threads package was built with the SPIN_RESCHED option, which it is by default, a **switch_pri** call will be done while waiting.) A spin lock is normally used to lock regions of short duration, when it is expected that any thread holding the lock will quickly release it.

PARAMETERS

p
: [pointer to in scalar] The spin lock to lock.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **spin_try_lock**, **spin_unlock**.

spin_try_lock

Function — Attempt to lock a spin lock

SYNOPSIS

```
boolean_t spin_try_lock
                (spin_lock_t*                                   p);
```

DESCRIPTION

The **spin_try_lock** function. makes a single attempt to lock *p*. The call does not block if the attempt to lock is unsuccessful.

PARAMETERS

p
: [pointer to in scalar] The spin lock to lock.

RETURN VALUE

TRUE
: if the lock is now locked to this thread

FALSE
: if the lock is still locked to some other thread

RELATED INFORMATION

Functions: **spin_lock**, **spin_unlock**.

spin_unlock

Function — Unlock a spin lock

SYNOPSIS

```
void spin_unlock
          (spin_lock_t*                                    p);
```

DESCRIPTION

The **spin_unlock** function unlocks the specified spin lock. This routine does not check to see if the lock was locked, nor that it was locked to this thread.

PARAMETERS

p
: [pointer to in scalar] The spin lock to be unlocked.

RETURN VALUE

None.

RELATED INFORMATION

Functions: **spin_try_lock**, **spin_lock**.

CHAPTER 4 Name Server

The name server provides a registry mapping service names to ports attached to the servers providing the named service.

netname_check_in

Function — Register a server

LIBRARY

libmach.a

#include <**servers/netname.h**>

SYNOPSIS

```
kern_return_t netname_check_in
                (mach_port_t                    server_port,
                 netname_name_t                   port_name,
                 mach_port_t                      signature,
                 mach_port_t                       port_id);
```

DESCRIPTION

The **netname_check_in** function registers the server receiving requests from port *port_id* that provides the service described / named by *port_name*. If the server is already known, *signature* must match that supplied when the server was previously registered. The *signature* value must be provided on all subsequent requests that affect this name to port mapping.

PARAMETERS

server_port
: [in scalar] Name server port

port_name
: [pointer to in array of *char*] String naming the service being provided

signature
: [in scalar] A port used to restrict who can re-register or de-register the server

port_id
: [in scalar] Port to the server

RETURN VALUE

NETNAME_SUCCESS
: The server was registered.

NETNAME_NOT_YOURS
An attempt was made to re-register a known server and the *signature* value did not match.

KERN_RESOURCE_SHORTAGE
Too many servers are being registered.

RELATED INFORMATION

Functions: **netname_check_out**, **netname_look_up**, **netname_version**.

netname_check_out

Function — De-register a server

LIBRARY

libmach.a

#include <**servers/netname.h**>

SYNOPSIS

```
kern_return_t netname_check_out
                    (mach_port_t                              server_port,
                    netname_name_t                            port_name,
                    mach_port_t                               signature);
```

DESCRIPTION

The **netname_check_out** function breaks the association between a service name and the registered port.

PARAMETERS

server_port
[in scalar] Name server. port

port_name
[pointer to in array of *char*] The service name to be de-registered.

signature
[in scalar] The value of the signature port used when registering the server.

RETURN VALUE

NETNAME_SUCCESS
The server was de-registered.

NETNAME_NOT_YOURS
An attempt was made to de-register a known server and the *signature* value did not match.

NETNAME_NOT_CHECKED_IN
No server is known by that name.

RELATED INFORMATION

Functions: **netname_check_in**, **netname_look_up**, **netname_version**.

netname_look_up

Function — Return a port to a named server

LIBRARY

libmach.a

#include <**servers/netname.h**>

SYNOPSIS

```
kern_return_t netname_look_up
                    (mach_port_t                         server_port,
                    netname_name_t                         host_name,
                    netname_name_t                         port_name,
                    mach_port_t*                             port_id);
```

DESCRIPTION

The **netname_look_up** function returns send rights to the port associated with a given service name.

PARAMETERS

server_port
: [in scalar] Name server port

host_name
: [pointer to in array of *char*] String specifying a particular host whose server is desired. A null string implies the current host. See the notes below.

port_name
: [pointer to in array of *char*] The name of the service desired.

port_id
: [out scalar] Send right to the port associated with the service

NOTES

The use of the *host_name* parameter depends on the name service involved.

The **snames** name server provides a single local name space only. The *host_name* parameter is ignored. All clients wishing to use the name space must have the port to the single **snames** server registered as their name server port.

The original Net Name server (part of the Net Message server) provides a set of per-node name spaces visible to one another. Clients on a node have as their registered name server port the port to the local name server. With this port they can look-up and check-in servers on their local node (by setting *host_name* to ""). With the *host_name* parameter to **netname_look_up**, they can locate servers on other nodes, including other nodes' name servers (checked-in as "NameServer").

RETURN VALUE

NETNAME_SUCCESS
The server port was returned.

NETNAME_NOT_CHECKED_IN
No service is known by the name (on the given host).

RELATED INFORMATION

Functions: **netname_check_in**, **netname_check_out**, **netname_version**.

netname_version

Function — Return a version string describing the name server

LIBRARY

libmach.a

#include <**servers/netname.h**>

SYNOPSIS

```
kern_return_t netname_version
                (mach_port_t                         server_port,
                netname_name_t                           version);
```

DESCRIPTION

The **netname_version** function returns a string naming which name server and which version is responding to *server_port*.

PARAMETERS

server_port
[in scalar] Name server port

version
[out array of *char*] Version string

RETURN VALUE

KERN_SUCCESS
Version string returned

RELATED INFORMATION

Functions: **netname_check_in**, **netname_check_out**, **netname_look_up**.

CHAPTER 5 NetMemory Server

The netmemory server provides shared memory objects whose contents are maintained consistently when mapped by multiple hosts.

netmemory_cache

Function — Create a Mach memory object from a netmemory object

LIBRARY

libmach.a

Not declared anywhere.

SYNOPSIS

```
kern_return_t netmemory_cache
               (mach_port_t                    netmemory_server,
                mach_port_t                    netmemory_object,
                mach_port_t*                   memory_object);
```

DESCRIPTION

The **netmemory_cache** function creates a Mach memory object on the local host given a netmemory object. The resulting memory object is suitable as a parameter to **vm_map**. The external memory manager for the resulting memory object is the local netmemory server which will co-ordinate with the other netmemory servers to consistently maintain the underlying netmemory object.

PARAMETERS

netmemory_server
[in scalar] Request port to the local netmemory server.

netmemory_object
[in scalar] Port representing the netmemory object

memory_object
[out scalar] Mach memory object suitable for **vm_map**

RETURN VALUE

NETMEMORY_SUCCESS
Operation succeeded

NETMEMORY_RESOURCE
The server could not allocate sufficient resources

NETMEMORY_BAD_PARAMETER
Invalid parameter supplied

KERN_FAILURE
netmemory_server does not name a known service.

RELATED INFORMATION

Functions: **netmemory_create**, **netmemory_destroy**.

netmemory_create

Function — Create a netmemory object

LIBRARY

libmach.a

Not declared anywhere.

SYNOPSIS

```
kern_return_t netmemory_create
        (mach_port_t                    netmemory_server,
         vm_size_t                           object_size,
         mach_port_t*                   netmemory_object,
         mach_port_t*                 netmemory_control);
```

DESCRIPTION

The **netmemory_create** function creates a netmemory object. The result is two ports: a *netmemory_control* port used for control operations upon the netmemory object (namely, object deletion) and a *netmemory_object* port which names the object for the **netmemory_cache** operation. Note that **netmemory_cache** must be invoked upon this *netmemory_object* port on each host to obtain a valid Mach memory object for use with **vm_map**.

PARAMETERS

netmemory_server
: [in scalar] Request port to the netmemory server.

object_size
: [in scalar] Size of the object in bytes

netmemory_object
: [out scalar] Port representing the netmemory object

netmemory_control
: [out scalar] Port used for control operations on the netmemory object

RETURN VALUE

NETMEMORY_SUCCESS
: Operation succeeded

NETMEMORY_RESOURCE
The server could not allocate sufficient resources

NETMEMORY_BAD_PARAMETER
Invalid parameter supplied

KERN_FAILURE
netmemory_server does not name a known service.

RELATED INFORMATION

Functions: **netmemory_cache**, **netmemory_destroy**.

netmemory_destroy

Function — Destroy a netmemory object

LIBRARY

libmach.a

Not declared anywhere.

SYNOPSIS

```
kern_return_t netmemory_destroy
                (mach_port_t                              netmemory_control);
```

DESCRIPTION

The **netmemory_destroy** function destroys the netmemory object.

PARAMETERS

netmemory_control
[in scalar] Port used for control operations on the netmemory object

RETURN VALUE

NETMEMORY_SUCCESS
Operation succeeded

KERN_FAILURE
netmemory_control does not name a valid object

RELATED INFORMATION

Functions: **netmemory_cache**, **netmemory_create**.

CHAPTER 6 Service Server

The service server provides a registry for the service server itself, the name server and the environment server. It exists so that the ports for these servers can be created at system initialization while the servers themselves are initialized later.

service_checkin

Function — Announce the presence of a base Mach server

LIBRARY

libmach.a

#include <**servers/service.h**>

SYNOPSIS

```
kern_return_t service_checkin
                 (mach_port_t          service_request,
                  mach_port_t          service_desired,
                  mach_port_t*         service_granted);
```

DESCRIPTION

The **service_checkin** function registers a base Mach server. The service request port, which up to this time was owned by the service server, is now owned by the requesting server. This call should be made only by the name and environment servers.

PARAMETERS

service_request
: [in scalar] Request port to the service server.

service_desired
: [in scalar] Send right to the port naming the server being registered.

service_granted
: [out scalar] Receive right to the port naming the server being registered.

RETURN VALUE

KERN_SUCCESS
: The requested service port was returned.

KERN_FAILURE
: *service_desired* does not name a known service or the service has already been registered.

RELATED INFORMATION

Functions: **service_waitfor**.

service_waitfor

Function — Wait for a base Mach server to be registered

LIBRARY

libmach.a

#include <**servers/service.h**>

SYNOPSIS

```
kern_return_t service_waitfor
                (mach_port_t                              service_request,
                 mach_port_t                              service_desired);
```

DESCRIPTION

The **service_waitfor** function suspends and does not return until the specified server checks-in to the service server.

PARAMETERS

service_request
[in scalar] Request port to the service server.

service_desired
[in scalar] Send right to the port naming the server desired.

RETURN VALUE

KERN_SUCCESS
The requested service has registered.

KERN_FAILURE
service_desired does not name a known service.

RELATED INFORMATION

Functions: **service_checkin**.

APPENDIX A C Language Functions

The ANSI C run-time function set includes functions that invoke operating system functionality (such as file operations). When writing a Mach server, though, especially when the server is the server that provides this operating system functionality, these functions will not be available.

If the server is being linked against **libmach.a**, which assumes the existence of the various Mach servers and a BSD server in many cases, the server would also be linked against the system's standard **libc.a** as well. Such a server may well also link against **libthreads.a**, which defines the C-threads package. This **libthreads** library must be linked before **libmach** or **libc**.

If, however, the server is intended to be a stand-alone server not dependent on these other servers, it would be linked against **libmach_sa.a** (and would not be linked with **libc.a**). In this case, the various C run-time functions are generally not available.

libmach_sa.a

Some C language functions of general utility that can be implemented without additional server support are provided in **libmach_sa.a** and listed here.

The following string functions are provided, exactly as in ANSI/K&R C:

bcopy **blkclr** **bzero** **memcpy** **strcatstrcmp**
strcpy **strlen** **strncpy**

The following variables are defined by **crt0.o** (in **libmach_sa.a**):

environ **errno**

The following C functions in **libmach_sa.a**, because of the nature of the stand-alone environment, differ from their normal counterparts as follows:

_start

The **_start** function performs C run-time, C thread, MIG and Mach related task start-up functions. This call occurs automatically when a task starts.

exit

The **exit** function terminates the calling task. It is equivalent to **task_terminate** (**task_self**()).

longjmp

The **libmach_sa longjmp** function differs from its C counterpart in that it does not manipulate signal mask state.

setjmp

The **libmach_sa setjmp** function differs from its C counterpart in that it does not manipulate signal mask state.

libthreads.a

Either a stand-alone server or a dependent server may link against **libthreads.a**. Beside threads themselves, the threads library also provides the following C language area related functions, redefined to properly handle multiple threads:

free **malloc** **realloc**

libmach.a

In general, **libmach.a** does not define any C language functions, assuming the existence of **libc.a**. A small handful of functions are defined or redefined as listed here.

atoh

This additional function, in the spirit of **atoi**, converts a hexadecimal string of characters digits (0 to 9, A to F and a to f) into a binary integer.

brk

This function is not implemented.

fork

The **fork** function is extended to call **mach_init** in the child process.

sbrk

This function is defined purely in terms of **vm_allocate**.

vfork

vfork is redefined to be the same as **fork**.

APPENDIX B Data Structure Definitions

This appendix discusses the specifics of the various structures used as a part of the server's various interfaces. This appendix does not discuss all of the various data types used by the server's interfaces, only the fields of the various structures used.

mig_reply_header

Structure — Defines the true type of information passed in and out of **mach_msg_server**

FILE

<**mach/mig_errors.h**>

SYNOPSIS

```
typedef struct
{
            mach_msg_header_t                 Head;
            mach_msg_type_t                   RetCodeType;
            kern_return_t                     RetCode;
} mig_reply_header_t;
```

DESCRIPTION

The **mig_reply_header** structure defines the format of the data interface between **mach_msg_server** and the various MIG generated servers it calls.

FIELDS

Head
The actual Mach IPC message

RetCodeType
Not used

RetCode
A return code to **mach_msg_server**, indicating the disposition of the return message. Refer to the *Server Writer's Guide* for a detailed explanation.

RELATED INFORMATION

Functions: **mach_msg_server.**

Data structures: **mach_msg_header.**

APPENDIX C Error Return Values

This appendix lists the various kernel return values.

An error code has the following format:

- system code (6 bits). The **err_get_system** (*err*) macro extracts this field.
- subsystem code (12 bits). The **err_get_sub** (*err*) macro extracts this field.
- error code (14 bits). The **err_get_code** (*err*) macro extracts this field.

The various system codes are:

- *err_kern* —kernel
- *err_us* — user space library
- *err_server*— user space servers
- *err_mach_ipc* — Mach-IPC errors
- *err_local* — user defined errors

A typical user error code definition would be:

> #define SOMETHING_WRONG *err_local* | **err_sub** (13) | 1

NETMEMORY_BAD_PARAMETER

Invalid parameter supplied

NETMEMORY_RESOURCE

The netmemory server could not allocate sufficient resources

NETMEMORY_SUCCESS

Operation succeeded

NETNAME_NOT_CHECKED_IN

No server is known by the given name.

NETNAME_NOT_YOURS

An attempt was made to change the registration of a server and the supplied *signature* value did not match.

NETNAME_SUCCESS

The name server operation was successful.

APPENDIX D Index

C Language Functions 95
C Thread Functions. 31
Data Structure Definitions 99
Error Return Values 101
Index 103
Interface Descriptions 1
Interface Types 2
Introduction. 1
Library Support Functions 5
MACH_PORT_VALID 6
Name Server 77
NetMemory Server 85
Parameter Types 3
Service Server 91
Special Forms 3
condition_alloc 32
condition_broadcast 33
condition_clear 34
condition_free 35
condition_init 36
condition_name 37
condition_set_name 38
condition_signal 39
condition_wait. 40
cthread_count 42
cthread_data 43
cthread_detach 44
cthread_exit. 45
cthread_fork 46
cthread_init 47
cthread_join 48
cthread_kernel_limit 49
cthread_limit 50
cthread_mach_msg 51
cthread_msg_active 53
cthread_msg_busy 54
cthread_name 55
cthread_self 56
cthread_set_data. 57
cthread_set_kernel_limit 58
cthread_set_limit 59
cthread_set_name. 60
cthread_stack_size 61
cthread_unwire. 62
cthread_wire. 63
cthread_yield 64
environment_port. 7
libmach.a 96
libmach_sa.a. 95
libthreads.a. 96
mach_device_server_port 8
mach_error. 9
mach_error_string 10
mach_error_type 11
mach_init 12
mach_msg_destroy 13

mach_msg_server 14
mach_privileged_host_port 16
mach_task_self. 17
mig_dealloc_reply_port 18
mig_get_reply_port 19
mig_init 20
mig_reply_header 100
mig_reply_setup. 21
mig_strncpy 23
mutex_alloc 65
mutex_clear 66
mutex_free 67
mutex_init 68
mutex_lock. 69
mutex_name 70
mutex_set_name 71
mutex_try_lock 72
mutex_unlock. 73
name_server_port. 24
netmemory_cache 86
netmemory_create 88
netmemory_destroy 90
netname_check_in 78
netname_check_out 80
netname_look_up 82
netname_version 84
quit 25
round_page. 26
service_checkin 92
service_port 27
service_waitfor. 93
slot_name 28
spin_lock 74
spin_try_lock 75
spin_unlock 76
trunc_page 29
vm_page_size. 30