MACH Kernel Interface Manual

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

Editor: Mary R. Thompson

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

Version of:
13 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.

1. Processor allocation primitives

1.1. Concepts

The processor allocation interface supports control over processors by user-mode programs. Application areas include gang scheduling and speedup measurement; both of these require allocating a specific number of processors to an application for the exclusive use of that application.

This interface introduces three new entities in addition to those already exported by the MACH kernel:

- **processor** - This corresponds to a hardware processor. Internally it corresponds to a data structure containing a local run queue. There is a distinguished processor, known as the master (until the kernelized system gets rid of it). There is a privileged port exported by the processor object to privileged servers which allows control of the physical processor.
- **processor_set** - This is a set of processors. Internally it corresponds to a data structure containing a "global" run queue and an idle queue. There is a distinguished set known as the default set to which all tasks, threads, and processors are initially assigned. Two ports are exported by a processor_set, a name port for obtaining information about the set, and an control port for performing operations on it. Neither of these ports are privileged.
- **host -** This represents the host. There are two ports exported by this object: a non-privileged port, `host`, for information queries, and a privileged port, `host_priv`, port that grants the right to manipulate physical resources. Other resource operations (e.g. making memory non-pageable) will be added in the future. The non-privileged port should be used to name the host.

Every processor is always assigned to exactly one processor set; a processor only executes threads that are assigned to its processor set. Every thread is always assigned to exactly one processor set, and only executes on processors assigned to that processor set[1]. Each task is also assigned to a processor set, but this assignment is only for the purposes of determining the initial assignment of newly created tasks and threads; tasks inherit their initial assignment from their parent, and threads inherit their initial assignment from the task that contains them. These assignments may be changed subsequently.

The host concept is introduced to isolate authentication concerns from the processor allocation interface. The host_priv and processor ports are privileged and only available to privileged servers. The kernel only provides allocation mechanisms for processors; policy is the responsibility of the server(s). In addition the servers may understand more about the topology of the machine (e.g. clustering of processors) than the kernel (the kernel makes no distinctions among processors internally). Processor sets are not privileged, and are intended to form a basis for the interfaces exported by the privileged servers. (Note that non-privileged users cannot obtain ports for the processors assigned to user-created processor sets even though they have the processor set control port.) The use of unprivileged processor sets exported by the kernel allows users to do scheduling (i.e. task and thread assignment) directly without having any resource control privileges.

This interface will exist on all machines, even uniprocessors; a kernel configuration switch deletes the code required to support processor allocation internally and returns failure codes. On a uniprocessor, the only processor set that exists is the default processor set; its control port is privileged and not available to most users. The calls to retrieve information about the processors and the default processor set are

---

[1]Unix[tm] system calls are an exception; they may force a thread to the master processor

useful even on such machines.

1.2. Functionality

This interface supports partitioning of the processing capability of a multiprocessor among applications in a fixed fashion. The granularity of this division is at the processor level. Processor sets are assembled by assigning processors to sets; a processor assigned to a set will only run threads that have been assigned to that set. Binding threads to individual processors is accomplished by creating processor sets containing exactly one processor. The master processor must always be assigned to the default set. Future versions of MACH intend to remove the need for a master processor. At that time the the default set must always be assigned at least one processor.

Binding of threads to individual processors can be achieved by assigning the threads to processor sets that contain exactly one processor. A kernel configuration switch is available to keep threads on the same processor in most situations where at most *n* threads are assigned to a processor set to which at least *n* processors are assigned.

All of the primitives except two are implemented via the ipc interface, and are therefore available over the network. This allows a long-term scheduler for a multiprocessor to reside on another machine. The two primitives that are implemented as traps are `host_self` and `host_priv_self`. The latter is only a trap pending implementation of an authorization/authentication mechanisms.

1.3. Uses

- Gang Scheduling - A gang scheduler can be written externally to the kernel. The scheduler can be implemented by shuffling processors among processor sets, thus avoiding any interaction with application use of task and thread control primitives.
- Speedup Measurement - A single processor set of *k* processors is sufficient to measure the performance of a parallel application on that many processors. The application may have more than *k* threads.
- User-Mode Scheduling - Assigning a single processor to each of several processor sets allows user applications complete control over the threads that execute on these processors. In addition to the **assign** primitives, the existing **suspend** and **resume** primitives are of potential utility.
- Application binding - Portions of an application can be bound to dedicated processors to optimize performance characteristics. Users may choose among many alternatives ranging from one processor set with all the dedicated processors and corresponding threads to one processor set for each dedicated processor.
- Load Balancing for Non-Uniform Memory Access (NUMA) Multiprocessors - Assigning each cluster of processors (i.e. processors with identical memory access characteristics) to a separate processor_set allows the kernel to perform load balancing only within clusters. A user-mode load balancer can then perform load balancing across clusters. Additional primitives to make load information available efficiently to support this are under design.

It should be noted that Speedup Measurement and Application Binding of Ada applications will usually require assigning a group of *k* threads to *n* processors, where $k > n > 1$, for best performance. This is due to the synchronous nature of Ada's rendezvous, and is also true of applications structured using remote procedure call. This functionality is not supported by a simpler interface that can only bind threads to dedicated processors.

host_self

```
#include <mach.h>

host_t host_self()

host_priv_t host_priv_self()
```

Description

`host_self` returns send rights to the host port for the host on which the call is executed. This port can only be used to obtain information about the host.

`host_priv_self` returns send rights to the privileged host port for the host on which the call is executed. This port is used to control physical resources on that host. It is needed for the operations that wire-down memory pagaes and the call that returns the processor ports which in turn allow control of specific processors. Currently, this port is only returned if the caller is the Unix super-user. Otherwise `PORT_NULL` is returned.

See Also

`host_processors`, `host_info`, `host_kernel_version`

Notes

Availability limited. `host_priv_self` should be replaced by a real authentication mechanism.

host_processors

```
#include <mach.h>

kern_return_t host_processors(host_priv, processor_list, processor_count)
    host_priv_t          host_priv;
    processor_array_t    *processor_list;        /* out, ptr to array */
    int                  *processor_count;       /* out */
```

Arguments

`host_priv` Privileged host port for the desired host.

`processor_list` The set of processors existing on `host_priv`, no particular ordering is guaranteed.

`processor_count`
The number of threads in the `processor_list`.

Description

`host_processors` gets send rights to the processor port for each processor existing on `host_priv`. This is the privileged port that allows its holder to control a processor. `processor_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`
`host_priv` is not a privileged host port.

`KERN_INVALID_ADDRESS`
`processor_count` points to inaccessible memory.

See Also

`processor_start`, `processor_exit`, `processor_info`, `processor_control`

Notes

Availability limited.

host_processor_sets

```
#include <mach.h>

kern_return_t
host_processor_sets(host, processor_set_list, processor_set_count)
    host_t                host;
    processor_set_array_t  *processor_set_list; /* out, ptr to array */
    int                    *processor_set_count;        /* out */
```

Arguments

host
: The host for which the list of processor sets is requested. Either the host port or the privileged host port may be used.

processor_set_list
: The set of processor_sets currently existing on host; no particular ordering is guaranteed.

processor_set_count
: The number of processor_sets in the processor_set_list.

Description

host_processor_sets gets send rights to the name port for each processor_set currently assigned to host. host_processor_set_priv can be used to obtain the object ports from these if desired. processor_set_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
: host is not a host.

Notes

Availability limited.

See Also

host_processor_set_priv, processor_set_create, processor_set_tasks, processor_set_threads

host_processor_set_priv

```
#include <mach.h>

kern_return_t host_processor_set_priv(host_priv, set_name, set);
        host_priv_t             host_priv;
        processor_set_name_t    set_name;
        processor_set_t         *set;
```

Arguments

host_priv	The privileged host port for the host on which this processor set resides.
set_name	The name port for this set.
set	Returns the control port for this set.

Description

This call allows a privileged application to obtain the control port for an existing processor set from its name port. The privileged host port is required.

Notes

Availability limited.

See Also

host_ports, processor_set_default, processor_set_create

host_info

```
#include <mach.h>

/* the definition of host_info_t from mach.h - sys/host_info.h is */

typedef int     *host_info_t;           /* variable length array of int */

/* two  interpretations of info are: */

   struct host_basic_info {
        int             max_cpus;       /* maximum possible cpus for
                                           which kernel is configured */
        int             avail_cpus;     /* number of cpus now available */
        vm_size_t       memory_size;    /* size of memory in bytes */
        cpu_type_t      cpu_type;       /* cpu type */
        cpu_subtype_t   cpu_subtype;    /* cpu subtype */
   };
typedef struct host_basic_info          *host_basic_info_t;


struct host_sched_info {
        int             min_timeout;    /* minimum timeout in milliseconds */
        int             min_quantum;    /* minimum quantum in milliseconds */
};
typedef struct host_sched_info *host_sched_info_t


kern_return_t host_info(host, flavor, host_info, host_infoCnt)
        host_t          host;
        int             flavor;
        host_info_t     host_info;      /* in and out */
        unsigned int    *host_infoCnt;  /* in and out */
```

Arguments

`host`	The host for which information is to be obtained
`flavor`	The type of statistics that are wanted. Currently `HOST_BASIC_INFO`, `HOST_PROCESSOR_SLOTS`, and `HOST_SCHED_INFO` are implemented.
`host_info`	Statistics about the host specified by `host`.
`host_infoCnt`	Size of the info structure. Should be `HOST_BASIC_INFO_COUNT` for `HOST_BASIC_INFO`. Should be the max number of cpus returned by `HOST_BASIC_INFO` for `HOST_PROCESSOR_SLOTS`. Should be `HOST_SCHED_INFO_COUNT` for `HOST_SCHED_INFO`.

Description

Returns the selected information array for a host, as specified by `flavor`. `host_info` is an array of integers that is supplied by the caller, and filled with specified information. `host_infoCnt` is supplied as the maximum number of integers in `host_info`. On return, it contains the actual number of integers in `host_info`. The host may be specified by either the host port or the privileged host port.

Basic information is defined by `HOST_BASIC_INFO`. The size of this information is defined by `HOST_BASIC_INFO_COUNT`. Processor slots of the active (available) processors is defined by `HOST_PROCESSOR_SLOTS`. The size of this information should be obtained from the `max_cpus` field of the structure returned by `HOST_BASIC_INFO`. Additional information of interest to schedulers is defined by `HOST_SCHED_INFO`. The size of this information is defined by `HOST_SCHED_INFO_COUNT`.

Returns

KERN_SUCCESS The call succeeded.

KERN_INVALID_ARGUMENT
: host is not a host or flavor is not recognized.

MIG_ARRAY_TOO_LARGE
: Returned info array is too large for host_info. host_info is filled as much as possible. host_infoCnt is set to the number of elements that would be returned if there were enough room.

Notes

Availability limited. Systems without this call support a host_info call with an incompatible calling sequence.

See Also

host_ports, host_kernel_version, host_processors, processor_info

host_kernel_version

```
#include <mach.h>

kern_return_t host_kernel_version(host, version)
        host_t host;
        kernel_version_t *version;              /* out */
```

Arguments

`host` The host for which information is being requested. Either the host port or the privileged host port may be used.

`version` Character string describing the kernel version executing on `host`.

Description

`host_kernel_version` returns the version string compiled into the kernel executing on `host` at the time it was built. This describes the version of the kernel. The constant `KERNEL_VERSION_MAX` should be used to dimension storage for the returned string if the `kernel_version_t` declaration is not used.

Returns

`KERN_SUCCESS` The call succeeded.

`KERN_INVALID_ARGUMENT`
`host` was not a host.

`KERN_INVALID_ADDRESS`
`version` points to inaccessible memory.

Notes

Availability limited.

See Also

`host_info`, `host_processors`, `host_ports`, `processor_info`

processor_assign

```
#include <mach.h>

kern_return_t processor_assign(processor, processor_set, wait)
        processor_t     processor;
        processor_set_t processor_set;
        boolean_t       wait;

kern_return_t processor_get_assignment(processor, assigned_set_name)
        processor_t     processor;
        processor_set_name_t assigned_set_name;
```

Arguments

`processor` The processor to be assigned.

`processor_set` The processor set to assign it to.

`wait` Whether to wait for the assignment to complete.

`assigned_set_name`
Name port for processor set that processor is currently assigned to

Description

`processor_assign` assigns `processor` to the the set `processor_set`. After the assignment is completed, the processor only executes threads that are assigned to that processor set. Any previous assignment of the processor is nullified. The master processor cannot be reassigned. All processors take clock interrupts at all times. The wait argument indicates whether the caller should wait for the assignment to be completed or should return immediately. Dedicated kernel threads are used to perform processor assignment, so setting `wait` to `FALSE` allows assignment requests to be queued and performed faster, especially if the kernel has more than one dedicated internal thread for processor assignment. Redirection of other device interrupts away from processors assigned to other than the default processor set is machine-dependent. Intermediaries that interpose on ports must be sure to interpose on both ports involved in this call if they interpose on either.

`processor_get_assignment` Obtains the current assignment of a processor. The name port of the processor set is returned.

Returns

`KERN_SUCCESS` The assignment has been performed.

`KERN_INVALID_ARGUMENT`
`processor` is not a processor, or `processor_set` is not a processor_set on the same host as `processor`.

Notes

Availability limited.

See Also

`processor_set_create`, `processor_set_info`, `task_assign`, `thread_assign`, `host_processor_set_priv`

processor_control

```
#include <mach.h>

kern_return_t processor_start(processor)
        processor_t             processor;

kern_return_t processor_exit(processor)
        processor_t             processor;

kern_return_t processor_control(processor, cmd, count)
        processor_t             processor;
        int                     *cmd;  /* array of ints */
        int                     count;
```

Arguments

`processor`	Specifies the processor.
`cmd`	Contains the command to be applied to the processor.
`count`	Specifies the length of the command as a number of ints.

Description

Some multiprocessors may allow privileged software to control processors. The `processor_start`, `processor_exit`, and `processor_control` operations implement this. The interpretation of the command in `cmd` is machine dependent. A newly started processor is assigned to the default processor set. An exited processor is removed from the processor set to which it was assigned and ceases to be active.

Returns

`KERN_SUCCESS`	The operation was performed.
`KERN_FAILURE`	The operation was not performed. A likely reason is that it is not supported on this processor.
`KERN_INVALID_ARGUMENT`	`processor` is not a processor.
`KERN_INVALID_ADDRESS`	`data` points to inaccessible memory.

See Also

`processor_info`, `host_processors`

Notes

Availability limited. All of these operations are machine-dependent. They may do nothing. The ability to restart an exited processor is also machine-dependent.

processor_info

```
#include <mach.h>

/* the definition of processor_info_t from mach.h - sys/processor_info.h is */

typedef int     *processor_info_t;      /* variable length array of int */

/* one interpretation of info is */

   struct processor_basic_info {
        cpu_type_t      cpu_type;       /* cpu type */
        cpu_subtype_t   cpu_subtype;    /* cpu subtype */
        boolean_t       running;        /* is processor running? */
        int             slot_num;       /* slot number */
        boolean_t       is_master;      /* is this the master processor */
   };
typedef struct processor_basic_info             *processor_basic_info_t;



kern_return_t
processor_info(processor, flavor, host, processor_info, processor_infoCnt)
        processor_t             processor;
        int                     flavor;
        host_t                  *host;
        processor_info_t        processor_info; /* in and out */
        unsigned int            *processor_infoCnt;     /* in and out */
```

Arguments

`processor` The processor for which information is to be obtained

`flavor` The type of information that is wanted. Currently only `PROCESSOR_BASIC_INFO` is implemented.

`host` The host on which the processor resides. This is the non-privileged host port.

`processor_info` Information about the processor specified by `processor`.

`processor_infoCnt`
Size of the info structure. Should be `PROCESSOR_BASIC_INFO_COUNT` for flavor `PROCESSOR_BASIC_INFO`.

Description

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

Basic information is defined by `PROCESSOR_BASIC_INFO`. The size of this information is defined by `PROCESSOR_BASIC_INFO_COUNT`. Machines which require more configuration information beyond the slot number are expected to define additional (machine-dependent) flavors.

Returns

`KERN_SUCCESS` The call succeeded.

`KERN_INVALID_ARGUMENT`
: `processor` is not a processor or `flavor` is not recognized.

`MIG_ARRAY_TOO_LARGE`
: Returned info array is too large for `processor_info`. `processor_info` is filled as much as possible. `processor_infoCnt` is set to the number of elements that would be returned if there were enough room.

Notes

Availability limited.

See Also

`processor_start`, `processor_exit`, `processor_control`, `host_processors` `host_info`

processor_set_create

```
#include <mach.h>

kern_return_t processor_set_create(host, new_set, new_name)
        host_t                  host;
        processor_set_t         *new_set;
        processor_set_name_t    *new_name;
```

Arguments

`host`	The host on which the new set is to be created. Either the host port or the privileged host port may be used.
`new_set`	Port used for performing operations on the new set.
`new_name`	Port used to identify the new set and obtain information about it.

Description

`processor_set_create` creates a new processor set and returns the two ports associated with it. The port returned in `new_set` is the actual port representing the set. It is used to perform operations such as assigning processors, tasks, or threads. The port returned in `new_name` identifies the set, and is used to obtain information about the set.

Returns

`KERN_SUCCESS`	The call succeeded.
`KERN_INVALID_ARGUMENT`	`host` was not a host.
`KERN_INVALID_ADDRESS`	`new_set` and/or `new_name` point to inaccessible memory.
`KERN_FAILURE`	The operating system does not support processor allocation.

Notes

Availability limited.

See Also

`processor_set_destroy`, `processor_set_info`, `processor_assign`, `task_assign`, `thread_assign`

processor_set_default

```
#include <mach.h>

kern_return_t processor_set_default(host, default_set);
        host_t                  host;
        processor_set_t         *default_set;
```

Arguments

`host`	Specifies the host whose default processor set is requested. Either the host port or the privileged host port may be used.
`default_set`	Returns the name port for the default processor set.

Description

The default processor set is used by all threads, tasks, and processors that are not explicitly assigned to other sets. `processor_set_default` returns a port that can be used to obtain information about this set (e.g. how many threads are assigned to it). This port cannot be used to perform operations on that set.

Notes

Availability limited.

See Also

`processor_set_info`, `thread_assign`, `task_assign`

processor_set_destroy

```
#include <mach.h>

kern_return_t processor_set_destroy(processor_set)
        processor_set_t         processor_set;
```

Arguments

processor_set — Specifies the processor_set to be exited.

Description

Destroys the specified processor set. Any assigned processors, tasks, or threads are reassigned to the default set. The object port for the processor set is required (not the name port). The default processor set cannot be destroyed.

Returns

KERN_SUCCESS — The set was destroyed.

KERN_FAILURE — An attempt was made to destroy the default processor set, or the operating system does not support processor allocation.

KERN_INVALID_ARGUMENT — processor_set is not a processor set.

Notes

Availability limited.

See Also

processor_set_create, processor_assign, task_assign, thread_assign

processor_set_info

```
#include <mach.h>

/* the definition of processor_set_info_ from mach/processor_info.h  */

typedef int     *processor_set_info_t;  /* variable length array of int */

/* one interpretation of info is */

   struct processor_set_basic_info {
        int             processor_count;        /* number of processors */
        int             task_count;             /* number of tasks */
        int             thread_count;           /* number of threads */
        int             load_average;           /* scaled load average */
        int             mach_factor;            /* scaled mach factor */
   };
typedef struct processor_set_basic_info         *processor_set_basic_info_t;

/* another interpretation of info is */

struct processor_set_sched_info {
        int             policies;       /* allowed policies */
        int             max_priority;   /* max priority for new threads */
};

typedef struct processor_set_sched_info *processor_set_sched_info_t;


kern_return_t
processor_set_info(processor_set, flavor, host, processor_set_info,
    processor_set_infoCnt)
        processor_set_name_t    processor_set;
        int                     flavor;
        host_t                  *host;
        processor_set_info_t    processor_set_info;     /* in and out */
        unsigned int            *processor_set_infoCnt; /* in and out */
```

Arguments

`processor_set`
: The processor_set for which information is to be obtained. Either the processor set name port or the processor set control port may be used.

`flavor`
: The type of information that is wanted.

`host`
: The host on which the processor set resides. This is the non-privileged host port.

`processor_set_info`
: Information about the processor set specified by `processor_set`.

`processor_set_infoCnt`
: Size of the info structure. Should be `PROCESSOR_SET_BASIC_INFO_COUNT` for flavor `PROCESSOR_SET_BASIC_INFO` , and `PROCESSOR_SET_SCHED_INFO_COUNT` for flavor `PROCESSOR_SET_SCHED_INFO`.

Description

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

Basic information is defined by `PROCESSOR_SET_BASIC_INFO`. The size of this information is defined by `PROCESSOR_SET_BASIC_INFO_COUNT`. The `load_average` and `mach_factor` arguments are scaled by the constant `LOAD_SCALE` (i.e. the integer value returned is the actual value multiplied by `LOAD_SCALE`.

Scheduling information is defined by `PROCESSOR_SET_SCHED_INFO`. The size of this information is given by `PROCESSOR_SET_SCHED_INFO_COUNT`. Some machines may define machine-dependent information flavors.

Returns

`KERN_SUCCESS` The call succeeded.

`KERN_INVALID_ARGUMENT`
: `processor_set` is not a processor set or `flavor` is not recognized.

`MIG_ARRAY_TOO_LARGE`
: Returned info array is too large for `processor_set_info`. `processor_set_info` is filled as much as possible. `processor_set_infoCnt` is set to the number of elements that would be returned if there were enough room.

Notes

Availability limited.

See Also

`processor_set_create`, `processor_set_default`, `processor_assign`, `task_assign`, `thread_assign`

processor_set_tasks

```
#include <mach.h>

kern_return_t processor_set_tasks(processor_set, task_list, task_count)
        processor_set_t processor_set;
        task_array_t    *task_list;     /* out, ptr to array */
        int             *task_count;    /* out */
```

Arguments

`processor_set`	The processor_set to be affected.
`task_list`	The set of tasks currently assigned to `processor_set`; no particular ordering is guaranteed.
`task_count`	The number of tasks in the `task_list.`

Description

`processor_set_tasks` gets send rights to the kernel port for each task currently assigned to `processor_set.` `task_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`	`processor_set` is not a processor_set.

Notes

Availability limited.

See Also

`task_assign, thread_assign, processor_set_threads`

processor_set_threads

```
#include <mach.h>

kern_return_t processor_set_threads(processor_set, thread_list, thread_count)
        processor_set_t processor_set;
        thread_array_t  *thread_list;   /* out, ptr to array */
        int             *thread_count;  /* out */
```

Arguments

`processor_set`	The processor_set to be affected.
`thread_list`	The set of threads currently assigned to `processor_set`; no particular ordering is guaranteed.
`thread_count`	The number of threads in the `thread_list`.

Description

`processor_set_threads` gets send rights to the kernel port for each thread currently assigned to `processor_set`. `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`	`processor_set` is not a processor_set.

Notes

Availability limited.

See Also

`thread_assign, thread_assign, processor_set_threads`

task_assign

```
#include <mach.h>

kern_return_t
task_assign(task, processor_set, assign_threads)
        task_t task;
        processor_set_t processor_set;
        boolean_t       assign_threads;

kern_return_t
task_assign_default(task, assign_threads)
        task_t          task;
        boolean_t       assign_threads;

kern_return_t
task_get_assignment(task, processor_set)
        task_t          task;
        processor_set_name_t    *processor_set;
```

Arguments

`task`	The task to be affected.
`processor_set`	The processor set to assign it to, or the processor set to which it is assigned.
`assign_threads`	Boolean indicating whether this assignment applies to existing threads in the task.

Description

`task_assign` assigns `task` the the set `processor_set.` This assignment is for the purposes of determining the initial assignment of newly created threads in `task.` Any previous assignment of the task is nullified. Existing threads within the task are also reassigned if `assign_threads` is TRUE. They are not affected if it is FALSE.

`task_assign_default` is a variant of `task_assign` that assigns the task to the default processor set on that task's host. This variant exists because the control port for the default processor set is privileged and not ususally available to users.

`task_get_assignment` returns the current assignment of the task.

Returns

`KERN_SUCCESS`	The assignment has been performed.
`KERN_INVALID_ARGUMENT`	`task` is not a task, or `processor_set` is not a processor_set on the same host as `task`.

Notes

Availability limited

See Also

`processor_set_create`, `processor_set_info`, `processor_assign`, `thread_assign`, `host_processor_set_priv`

thread_assign

```
#include <mach.h>

kern_return_t thread_assign(thread, processor_set)
        thread_t        thread;
        processor_set_t processor_set;

kern_return_t thread_assign_default(thread)
        thread_t        thread;

kern_return_t thread_get_assignment(thread, processor_set)
        thread_t                thread;
        processor_set_name_t    *processor_set;
```

Arguments

`thread` The thread to be assigned.

`processor_set` The processor set to assign it to.

Description

`thread_assign` assigns `thread` the the set `processor_set.` After the assignment is completed, the thread only executes on processors assigned to the designated processor set. If there are no such processors, then the thread is unable to execute. Any previous assignment of the thread is nullified. Unix system call compatibility code may temporarily force threads to execute on the master processor.

`thread_assign_default` is a variant of `thread_assign` that assigns the thread to the default processor set. This variant exists because the control port for the default processor set is privileged and therefore not available to most users.

`thread_get_assignment` returns the name of the processor set to which the thread is currently assigned. This port can only be used to obtain information about the processor set.

Returns

`KERN_SUCCESS` The assignment has been performed.

`KERN_INVALID_ARGUMENT`
`thread` is not a thread, or `processor_set` is not a processor_set on the same host as `thread.`

`KERN_INVALID_ADDRESS`
`processor_set` points to inaccessible memory (thread_get_assignment only).

Notes

Availability limited.

See Also

`processor_set_create`, `processor_set_info`, `processor_assign`, `task_assign`, `host_processor_set_priv`

2. Scheduling primitives

2.1. Introduction

This section of the manual describes primitives that control three aspects of thread scheduling.

1. **Priority -** These primitives export priorities for individual threads. The notion of scheduling policies is introduced, along with support for fixed-priority threads. threads.
2. **Handoff -** Handoff Scheduling (specify thread to run next) and related mechanisms.
3. **Wiring -** Lock data and threads into memory to prevent paging and swapping.

The priority primitives are connected with the processor set managment primitives and rely on presentation of processor set control ports to enforce protection. The overriding model is that a task that has rights to a processor set control port may exercise complete control over scheduling on that processor set.

2.2. Priority

The priority primitives:

1. Export priorities for individual threads to users.
2. Support fixed priorities for real-time and other uses.
3. Produce a clean interface for users as well as kernel clients.

Threads have both a priority and a maximum priority: priority cannot exceed maximum priority, but the maximum priority can be reset by presenting the appropriate processor_set object port. Since the default processor set's object port is privileged, ordinary users who do not do their own processor allocation cannot raise thread priorities above their initial maximum. Initial priority is inherited from the task at creation (of both threads and tasks), and the initial maximum priority is inherited from the processor set at thread creation.

Scheduling policy can be set on a per thread basis; the current implementation has two policies, time sharing and fixed priority. For fixed priority, a quantum can be specified (again on a per thread basis); this is the quantum that the thread will receive before being eligible for preemption at the same priority. A processor set may forbid scheduling policies other than time sharing.

The ranges of priority correspond to scheduler internals, but Unix interfaces will continue to deal with Unix priorities. Two notions of priority are supported; a base priority and a scheduled priority. The base priority ranges from 0 to 31 and is the priority assigned to the thread by the user or user-mode scheduler. The scheduled priority also has the same range; it is identical to the base priority for fixed priority threads, but for timesharing threads it is the base priority plus some increment derived from usage. In both cases low numbers are the highest priorities. The traditional Unix priority (range 0-127) can be obtained from these numbers (range 0-31) by multiplying by 4.

Preemption normally occurs when a thread becomes runnable at a higher priority than a currently running thread and there are no idle processors in the processor set. It may take as long as a (machine-dependent) clock interrupt period for a multiprocessor kernel to notice that preemption is needed (interprocessor interrupts are not used for preemption events). There is one major exception to this rule that delays preemption of time-sharing threads. The reason for this is that time-sharing threads have their priority recalculated as they run; if these new priorities were always put into effect immediately, the result would be additional (unneeded) context switches. To avoid this, running timesharing threads usually delay preemption due to this priority recalculation until the end of their current quantum (typically 1/10 of a second).

The policy for threads defaults to time sharing in the absence of explicit specification. Similarly, processor sets default to only allowing time sharing.

I/O drivers may prioritize requests placed by fixed priority threads according to the priorities of the threads; this behavior is machine-dependent. Any such prioritizing applies only to I/O explicitly requested by a fixed priority thread; implicit I/O (e.g. that requested by an external memory manager on behalf of a thread) will only be prioritized according to the thread that actually requested it (e.g. the thread in the external memory manager). As a result, systems that care about the priority of implicitly requested I/O should provide their own external memory managers to perform that I/O. The notion of priority may not be meaningful to some potential future policies (e.g. round-robin).

The timesharing policy must be allowed to simplify the semantics and implementation of the thread and task assign operations. Assignment of a thread whose policy is forbidden by the target processor set succeeds, but its policy is reset to timesharing; the policy for newly created threads similarly defaults to timesharing. This avoids the need to introduce a notion of default policy for processor sets and the additional logic needed to make sure the default policy is permitted and at least one policy is permitted at all times. Timesharing can still be avoided on user-created and managed processor sets by resetting the thread's policy after creation or assignment; a real-time scheduler will surely want to do this in any case.

When a thread is assigned to a processor set, its priorities (both actual and max) will be reduced if either is above the target set's maximum priority. In addition its scheduling policy will be reset to time sharing if the target processor set does not permit its current policy. Default values of priority and max priority are 12 for historical reasons.

`processor_set_max_priority` may be used as a boot-time mechanism to clear out high-level priorities for a real-time subsystem by indicating the max_priority for time sharing threads and setting change_threads to TRUE.

2.3. Handoff Scheduling

Handoff scheduling refers to the technique of transferring the processor from one thread to another with as little operating system involvement as possible. The kernel already uses this technique for message operations. The primitive `thread_switch` makes it and related operations available to the user. These primitives allow the user to take advantage of knowledge about which threads should or should not be run to influence operating system scheduling decisions without requiring the user to write a complete scheduler.

This call is independent of the priority manipulation calls; they are motivated by situations encountered in doing locking on multiprogrammed systems. `thread_switch` is a trap that operates on the thread that invokes it.

Two options to `thread_switch` are supported:

1. **WAIT** - option_time is a time period during which the current thread should be suspended internally. An internal mechanism will be used that cannot be released by thread_resume; thread_abort must be used to abort this wait. This option is designed for situations in which there is a significant minimum time for which the thread should be blocked.
2. **DEPRESS** - option_time is a time_period during which the current thread's priority should be reduced to the lowest possible value. Priority is restored by expiration of the timeout, the scheduler choosing to run the thread again, or a call to `thread_abort`. Experience with parallel applications at CMU indicates that a primitive that only context switches (and implicitly surrenders the remainder of the quantum) is not sufficient; two threads performing this operation can hog the processor to the exclusion of threads that should run. It is necessary to severely depress the priority of the threads that should not be run so that the other threads will be run even if their priority is below that of the waiting threads due to actions of the timesharing scheduler. The advantage of using depression over blocking is that an explicit unblock operation is not needed to resume the thread before the timeout expires. Implementors should be warned that large number of threads spinning through thread_switch or thread_switch with the DEPRESS option can effectively thrash a multiprocessor scheduler.

The specification of the next thread to run is designed to handle situations in which the user is keeping explicit or implicit scheduling information that can be taken advantage of by the kernel. An example of explicit information is tagging locks with the identity of the thread that holds them (this is easy to do if the hardware supports an atomic compare and swap if 0 operation). An example of implicit information is that in a functionally partitioned program with one thread per component, knowing the component that is being waited for immediately identified the thread that should run.

The current quantum is transferred to the new thread by `thread_switch` only if both thread and new_thread are time-sharing threads. If either or both are fixed-priority threads, the new_thread gets a new quantum.

Fixed priority threads do not get special treatment for any of these calls. This means that thread_switch will run the new thread regardless of its priority, and can be used to implement the implicit priority elevation required when a critical section is accessed by both low and high priority threads. Similarly, the calls involving depression operate identically on both fixed priority and time sharing threads.

2.4. Wiring

There are two wiring primitives: `vm_wire` and `thread_wire`. `vm_wire` allows the user to keep memory from pagefaulting on specified types of access. `thread_wire` causes a thread's internal kernel state to be non-pageable so that thread will always be able to execute immediately when run. The system has an overall limit on wired down memory; these calls fail if the requested wiring would exceed that limit.

NOTE: These calls have not been implemented yet.

task_priority

```
#include <mach.h>

kern_return_t task_priority(task, priority, change_threads)
        task_t          task;
        int             priority;
        boolean_t       change_threads;
```

Arguments

`task`	Task to set priority for.
`priority`	New priority.
`change_threads`	Change priority of existing threads if TRUE.

Description

The priority of a task is used only for creation of new threads; a new thread's priority is set to the enclosing task's priority. `task_priority` changes this task priority. It also sets the priorities of all threads in the task to this new priority if `change_threads` is TRUE. Existing threads are not affected otherwise. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

Returns

`KERN_SUCCESS`	The call succeeded.
`KERN_INVALID_ARGUMENT`	`task` is not a task, or `priority` is not a valid priority.
`KERN_FAILURE`	`change_threads` was TRUE and the attempt to change the priority of at least one existing thread failed because the new priority would have exceeded that thread's maximum priority.

Notes

Availability limited.

See Also

`thread_priority, processor_set_max_priority`

thread_priority

```
#include <mach.h>

kern_return_t thread_priority(thread, priority, set_max)
        thread_t        thread;
        int             priority;
        boolean_t       set_max

kern_return_t thread_max_priority(thread, processor_set, priority)
        thread_t        thread;
        processor_set_t processor_set;
        int             priority;
```

Arguments

`thread`	The thread whose priority is to be changed.
`priority`	The new priority to change it to.
`set_max`	Also set thread's maximum priority if TRUE.
`processor_set`	The control port for the processor set to which the thread is currently assigned.

Description

Threads have three priorities associated with them by the system, a `priority`, a `maximum priority`, and a `scheduled priority.` The scheduled priority is used to make scheduling decisions about the thread. It is determined from the priority by the policy (for timesharing, this means adding an increment derived from cpu usage). The priority can be set under user control, but may never exceed the maximum priority. Changing the maximum priority requires presentation of the control port for the thread's processor set; since the control port for the default processor set is privileged, users cannot raise their maximum priority to unfairly compete with other users on that set. Newly created threads obtain their priority from their task and their max priority from the thread.

`thread_priority` changes the priority and optionally the maximum priority of `thread.` Priorities range from 0 to 31, where lower numbers denote higher priorities. If the new priority is higher than the priority of the current thread, preemption may occur as a result of this call. The maximum priority of the thread is also set if `set_max` is TRUE. This call will fail if `priority` is greater than the current maximum priority of the thread. As a result, this call can only lower the value of a thread's maximum priority.

`thread_max_priority` changes the maximum priority of the thread. Because it requires presentation of the corresponding processor set port, this call can reset the maximum priority to any legal value.

Returns

`KERN_SUCCESS`	Operation completed successfully
`KERN_INVALID_ARGUMENT`	`thread` is not a thread, or `processor_set` is not a control port for a processor set, or `priority` is out of range (not in 0..31).
`KERN_FAILURE`	The requested operation would violate the thread's maximum priority (thread_priority) or the thread is not assigned to the processor set whose control port was presented.

Notes

Availability limited.

See Also

`thread_policy`, `task_priority`, `processor_set_priority`

thread_policy

```
#include <mach.h>

kern_return_t thread_policy(thread, policy, data)
        thread_t        thread;
        int             policy;
        int             data;
```

Arguments

thread	Thread to set policy for.
policy	Policy to set.
data	Policy-specific data.

Description

thread_policy changes the scheduling policy for thread to policy. data is policy-dependent scheduling information. There are currently two supported policies: POLICY_TIMESHARE and POLICY_FIXEDPRI defined in <mach/policy.h>; this file is included by mach.h. data is meaningless for timesharing, but is the quantum to be used (in milliseconds) for the fixed priority policy. To be meaningful, this quantum must be a multiple of the basic system quantum (min_quantum) which can be obtained from host_info. The system will always round up to the next multiple of the quantum.

Processor sets may restrict the allowed policies, so this call will fail if the processor set to which thread is currently assigned does not permit policy.

Returns

KERN_SUCCESS	The call succeeded.
KERN_INVALID_ARGUMENT	thread is not a thread, or policy is not a recognized policy.
KERN_FAILURE	The processor set to which thread is currently assigned does not permit policy.

Notes

Availability limited. Fixed priority not supported on all systems.

See Also

task_policy, processor_set_policy, host_info

processor_set_max_priority

```
#include <mach.h>

kern_return_t
processor_set_max_priority(processor_set, priority, change_threads)
        processor_set_t         task;
        int                     priority;
        boolean_t               change_threads;
```

Arguments

`processor_set` Processor set to set maximum priority for.

`priority` New priority.

`change_threads` Change maximum priority of existing threads if TRUE.

Description

The priority of a processor set is used only for newly created threads (thread's maximum priority is set to processor set's) and the assignment of threads to the set (thread's maximum priority is reduced if it exceeds the set's maximum priority, thread's priority is similarly reduced). `processor_set_max_priority` changes this priority. It also sets the maximum priority of all threads assigned to the processor set to this new priority if `change_threads` is TRUE. If this maximum priority is less than the priorities of any of these threads, their priorities will also be set to this new value.

Returns

`KERN_SUCCESS` The call succeeded.

`KERN_INVALID_ARGUMENT`
`task` is not a task, or `priority` is not a valid priority.

Notes

Availability limited. This call was referred to as `processor_set_priority` in some previous documentation.

See Also

`thread_priority, task_priority, thread_assign`

processor_set_policy_enable

```
#include <mach.h>

kern_return_t
processor_set_policy_enable(processor_set, policy)
        processor_set_t processor_set;
        int             policy;

kern_return_t
processor_set_policy_disable(processor_set, policy, change_threads)
        thread_t        thread;
        processor_set_t processor_set;
        int             change_threads;
```

Arguments

`processor_set`	The processor set whose allowed policies are to be changed.
`policy`	The policy to enable or disable
`change_threads`	Reset the policies of any threads with the newly-disallowed policy to timesharing.

Description

Processor sets may restrict the scheduling policies to be used for threads assigned to them. These two calls provide the mechanism for designating permitted and forbidden policies. The current set of permitted policies can be obtained from `processor_set_info`. Timesharing may not be forbidden by any processor_set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by the target processor set has its policy reset to timesharing. If the `change_threads` argument to `processor_set_policy_disable` is true, threads currently assigned to this processor set and using the newly disabled policy will have their policy reset to timesharing.

<mach/policy.h> contains the allowed policies; it is included by mach.h.

Returns

`KERN_SUCCESS`	Operation completed successfully
`KERN_INVALID_ARGUMENT`	`processor_set` is not a processor set, or `policy` is not a valid policy, or an attempt was made to disable timesharing.

Notes

Availability limited. Not all policies (e.g. fixed priority) are supported by all systems.

See Also

`thread_policy, task_policy`

thread_switch

```
#include <mach.h>

kern_return_t thread_switch(new_thread, option, time)
        thread_t        new_thread;
        int             option;
        int             time;
```

Arguments

`new_thread`	Thread to context switch to.
`option`	Specifies options associated with context switch.
`time`	Time duration for options
`thread`	Thread to be affected.

Description

`thread_switch` provides low-level access to the scheduler's context switching code. `new_thread` is a hint that implements handoff scheduling. The operating system will attempt to switch directly to the new thread (bypassing the normal logic that selects the next thread to run) if possible. Since this is a hint, it may be incorrect; it is ignored if it doesn't specify a thread on the same host as the current thread or if that thread can't be switched to (not runnable or already running on another processor). In this case, the normal logic to select the next thread to run is used; the current thread may continue running if there is no other appropriate thread to run.

Options for `option` are defined in <mach/thread_switch.h>. Three options are recognized: `SWITCH_OPTION_NONE` No options, the time argument is ignored. `SWITCH_OPTION_WAIT` The thread is blocked for the specified `time.` This can be aborted by `thread_abort.` `SWITCH_OPTION_DEPRESS` The thread's priority is depressed to the lowest possible value for `time.` This is aborted by `thread_abort`, or by the scheduler choosing to run the thread again. A consequence of this is that a depressed thread must be queued (on a run queue) waiting for the scheduler to choose it to run; once chosen, the depression is aborted. This depression is independent of operations that change the thread's priority (e.g. `thread_priority` will not abort the depression). The minimum time and units of time can be obtained as the min_timeout value from `host_info.`

`thread_switch` is an optimized trap that affects the current thread. `thread_depress_abort` is an rpc to the kernel that may affect any thread.

`thread_switch` is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The `new_thread` argument (handoff scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The `WAIT` option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly long. The `DEPRESS` option is used when the amount of time that must be waited is fairly short, especially when the identity of the thread that is being waited for is not known.

Users should beware of calling `thread_switch` with an invalid hint (e.g. `THREAD_NULL`) and no option. Because the time-sharing scheduler varies the priority of threads based on usage, this may result in a waste of cpu time if the thread that must be run is of lower priority. The use of the DEPRESS option in this situation is highly recommended.

thread_switch ignores policies. Users relying on the preemption semantics of a fixed time policy should be aware that thread_switch ignores these semantics; it will run the specified new_thread indepent of its priority and the priority of any other threads that could be run instead.

Returns

KERN_SUCCESS — The call succeeded. thread_restore_priority always succeeds, and does not have a defined return value as a result.

KERN_INVALID_ARGUMENT — thread is not a thread, or option is not a recognized option.

KERN_FAILURE — kern_depress_abort failed because the thread was not depressed.

Notes

Availability limited.

See Also

host_info

thread_wire

```
#include <mach.h>

kern_return_t thread_wire(host_priv, thread, wired)
        host_priv_t     host_priv;
        thread_t        thread;
        boolean_t       wired;
```

Arguments

`host_priv`	The privileged host port for the thread's host.
`thread`	The thread to be affected
`wired`	Make thread unswappable if TRUE, swappable if FALSE.

Description

`thread_wire` allows privileged to restrict the swappability of threads. A unswappable thread has its kernel stack wired (made non-pageable) so it cannot be swapped out. This counts against the limit of wired memory maintained by the kernel (see `vm_wire`). The number of pages that is consumed can be obtained from `vm_wire_statistics`.

Returns

`KERN_SUCCESS`	The call succeeded
`KERN_INVALID_ARGUMENT`	`host_priv` is not the privileged host port for `thread`'s host, or `thread` is not a thread.
`KERN_RESOURCE_SHORTAGE`	Some kernel resource limit, most likely that on the amount of memory that can be wired down, has been exceeded.

Notes

Statistics interface to obtain wire limit and count against that limit (`vm_wire_statistics`) not available yet. Availability limited.

See Also

`vm_wire`, `vm_wire_statistics`, `host_priv_self`

vm_wire

```
#include <mach.h>

kern_return_t vm_wire (host_priv, task, start, end, prot)
        host_priv_t     host_priv;
        task_t          task;
        vm_address_t    start,end;
        vm_prot_t       prot;
```

Arguments

`host_priv`	The privileged host port for the host on which task resides
`task`	The task whose memory is to be affected
`start,end`	First and last addresses of the memory region to be affected.
`prot`	Types of accesses that must not cause page faults.

Description

`vm_wire` allows privileged applications to control memory pageability. The semantics of a successful `vm_wire` operation are that memory in the range from `start` to `end` in `task` will not cause page faults for any accesses included in `prot`. Data memory can be made non-pageable (wired) with a `prot` argument of `VM_PROT_READ|VM_PROT_WRITE.` A special case is that `VM_PROT_NONE` makes the memory pageable. Machines with hardware restrictions on address aliasing (e.g. IBM PC/RT, HP-PA) may have to reload mappings to wired memory and flush caches if more than one virtual mapping corresponding to the same physical memory is used. The mappings will usually be reloaded from a fast software cache, but if this overhead is a problem aliasing of wired memory should be avoided on these architectures.

The kernel maintains an internal limit on how much memory may be wired to protect itself from attempts to wire all of physical memory or more. Attempting to wire more memory than this limit allows will fail. The limit is a limit on address space, so wiring shared memory twice counts against the limit twice.

Returns

`KERN_SUCCESS` The call succeeded.

`KERN_INVALID_ARGUMENT`
: `host_priv` is not the privileged host port for `task`'s host, or `task` is not a task, or `start` and `end` do not define a valid address range in `task`, or `prot` is not a valid memory protection.

`KERN_RESOURCE_SHORTAGE`
: Some kernel resource limit, most likely that on the amount of memory that can be wired down, has been exceeded.

`KERN_FAILURE` Some memory in the range from `start` to `end` does not exist.

`KERN_PROTECTION_FAILURE`
: Some memory in the range from `start` to `end` does not allow all accesses specified by `prot`

Notes

The limit on wired memory should discount for sharing. The kernel may choose to wire for write access even if that is not specified in `prot` causing unexpected copies to be made. This behavior can be avoided by setting the maximum protection on the memory to read-only. Statistics interface to obtain wire limit and count against that limit (`vm_wire_statistics`) not available yet. Availability limited.

See Also

`thread_wire`, `vm_wire_statistics`, `host_priv_self`

Table of Contents

1. Processor allocation primitives	**1**
1.1. Concepts	1
1.2. Functionality	2
1.3. Uses	2
2. Scheduling primitives	**24**
2.1. Introduction	24
2.2. Priority	24
2.3. Handoff Scheduling	25
2.4. Wiring	26

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.

```
[4]   host_t host_self()
```

```
[4]   host_priv_t host_priv_self()
```

```
[5]   kern_return_t host_processors(host_priv, processor_list, processor_count
          host_priv_t         host_priv;
          processor_array_t   *processor_list;        /* out, ptr to array */
          int                 *processor_count;       /* out */
```

```
[6]   kern_return_t
      host_processor_sets(host, processor_set_list, processor_set_count)
          host_t                 host;
          processor_set_array_t  *processor_set_list; /* out, ptr to array */
          int                    *processor_set_count;        /* out */
```

```
[7]   kern_return_t host_processor_set_priv(host_priv, set_name, set);
              host_priv_t            host_priv;
              processor_set_name_t   set_name;
              processor_set_t        *set;
```

```
[8]   kern_return_t host_info(host, flavor, host_info, host_infoCnt)
              host_t          host;
              int             flavor;
              host_info_t     host_info;      /* in and out */
              unsigned int    *host_infoCnt;  /* in and out */
```

```
[10]  kern_return_t host_kernel_version(host, version)
              host_t host;
              kernel_version_t *version;              /* out */
```

```
[11]  kern_return_t processor_assign(processor, processor_set, wait)
              processor_t     processor;
              processor_set_t processor_set;
              boolean_t       wait;
```

```
[11]  kern_return_t processor_get_assignment(processor, assigned_set_name)
              processor_t     processor;
              processor_set_name_t assigned_set_name;
```

```
[12]  kern_return_t processor_start(processor)
              processor_t             processor;
```

```
[12]  kern_return_t processor_exit(processor)
              processor_t             processor;
```

```
[12]  kern_return_t processor_control(processor, cmd, count)
              processor_t             processor;
              int                     *cmd;  /* array of ints */
              int                     count;
```

```
[13]  /* the definition of processor_info_t from mach.h - sys/processor_info.h

      typedef int     *processor_info_t;      /* variable length array of int

      /* one interpretation of info is */

         struct processor_basic_info {
              cpu_type_t      cpu_type;       /* cpu type */
              cpu_subtype_t   cpu_subtype;    /* cpu subtype */
              boolean_t       running;        /* is processor running? */
              int             slot_num;       /* slot number */
              boolean_t       is_master;      /* is this the master processor
         };
      typedef struct processor_basic_info             *processor_basic_info_t;


      kern_return_t
      processor_info(processor, flavor, host, processor_info, processor_infoCn
              processor_t             processor;
              int                     flavor;
              host_t                  *host;
              processor_info_t        processor_info; /* in and out */
              unsigned int            *processor_infoCnt;     /* in and out */
```

```
[15]  kern_return_t processor_set_create(host, new_set, new_name)
              host_t                  host;
              processor_set_t         *new_set;
              processor_set_name_t    *new_name;
```

```
[16]  kern_return_t processor_set_default(host, default_set);
              host_t                host;
              processor_set_t       *default_set;
```

```
[17]  kern_return_t processor_set_destroy(processor_set)
              processor_set_t       processor_set;
```

```
[18]  /* the definition of processor_set_info_ from mach/processor_info.h  */

      typedef int     *processor_set_info_t;  /* variable length array of int

      /* one interpretation of info is */

         struct processor_set_basic_info {
              int              processor_count;        /* number of processors
              int              task_count;             /* number of tasks */
              int              thread_count;           /* number of threads */
              int              load_average;           /* scaled load average *
              int              mach_factor;            /* scaled mach factor */
         };
      typedef struct processor_set_basic_info         *processor_set_basic_inf

      /* another interpretation of info is */

      struct processor_set_sched_info {
              int              policies;       /* allowed policies */
              int              max_priority;   /* max priority for new threads
      };

      typedef struct processor_set_sched_info *processor_set_sched_info_t;


      kern_return_t
      processor_set_info(processor_set, flavor, host, processor_set_info,
          processor_set_infoCnt)
              processor_set_name_t    processor_set;
              int                     flavor;
              host_t                  *host;
              processor_set_info_t    processor_set_info;     /* in and out */
              unsigned int            *processor_set_infoCnt; /* in and out */
```

```
[20]  kern_return_t processor_set_tasks(processor_set, task_list, task_count)
              processor_set_t processor_set;
              task_array_t    *task_list;     /* out, ptr to array */
              int             *task_count;    /* out */
```

```
[21]  kern_return_t processor_set_threads(processor_set, thread_list, thread_c
```

```
        processor_set_t processor_set;
        thread_array_t  *thread_list;   /* out, ptr to array */
        int             *thread_count;  /* out */
```

```
[22]  kern_return_t
      task_assign(task, processor_set, assign_threads)
              task_t task;
              processor_set_t processor_set;
              boolean_t       assign_threads;
```

```
[22]  kern_return_t
      task_assign_default(task, assign_threads)
              task_t          task;
              boolean_t       assign_threads;
```

```
[22]  kern_return_t
      task_get_assignment(task, processor_set)
              task_t          task;
              processor_set_name_t    *processor_set;
```

```
[23]  kern_return_t thread_assign(thread, processor_set)
              thread_t        thread;
              processor_set_t processor_set;
```

```
[23]  kern_return_t thread_assign_default(thread)
              thread_t        thread;
```

```
[23]  kern_return_t thread_get_assignment(thread, processor_set)
              thread_t                thread;
              processor_set_name_t    *processor_set;
```

```
[27]  kern_return_t task_priority(task, priority, change_threads)
              task_t          task;
              int             priority;
              boolean_t       change_threads;
```

```
[28]  kern_return_t thread_priority(thread, priority, set_max)
```

```
        thread_t        thread;
        int             priority;
        boolean_t       set_max
```

```
[28]  kern_return_t thread_max_priority(thread, processor_set, priority)
        thread_t        thread;
        processor_set_t processor_set;
        int             priority;
```

```
[30]  kern_return_t thread_policy(thread, policy, data)
        thread_t        thread;
        int             policy;
        int             data;
```

```
[31]  kern_return_t
      processor_set_max_priority(processor_set, priority, change_threads)
        processor_set_t         task;
        int                     priority;
        boolean_t               change_threads;
```

```
[32]  kern_return_t
      processor_set_policy_enable(processor_set, policy)
        processor_set_t processor_set;
        int             policy;
```

```
[32]  kern_return_t
      processor_set_policy_disable(processor_set, policy, change_threads)
        thread_t        thread;
        processor_set_t processor_set;
        int             change_threads;
```

```
[33]  kern_return_t thread_switch(new_thread, option, time)
        thread_t        new_thread;
        int             option;
        int             time;
```

```
[35]  kern_return_t thread_wire(host_priv, thread, wired)
        host_priv_t     host_priv;
        thread_t        thread;
        boolean_t       wired;
```

```
[36]  kern_return_t vm_wire (host_priv, task, start, end, prot)
              host_priv_t     host_priv;
              task_t          task;
              vm_address_t    start,end;
              vm_prot_t       prot;
```