Mach Processor Allocation Interface

DRAFT of 13 August 1989

David L. Black
Carnegie Mellon University

Introduction

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

Concepts

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).
- **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 object port for performing operations on it.
- **host** - This represents the host. There are two versions of this object: a non-privileged version for information queries, and a privileged version 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 and processor entities 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 users cannot obtain ports for the processors assigned to user-created processor sets.) The use of a kernel exported entity 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

---

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

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 useful even on such machines.

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 [to be replaced by: the default set must always be assigned at least one processor].

Primitives

The interface primitives are:

- Host operations:
  - host_t **host_self** -- Trap to return host port. This port can only be used for obtaining information about the host.
  - host_t **host_priv_self** -- Trap to return privileged host port. PORT_NULL is returned if invoker is not the super-user. (This trap may be replaced when we start to use a real authentication mechanism).
  - kern_return_t **host_processors**(host_priv, *processor_list, *processor_count) Return all of the processor ports for the given host. The ports are returned in an array; position of ports in the array need not correspond to slot numbers (and won’t on architectures with sparse processor numbering). The privileged host port (from host_priv_self()) must be used.
  - kern_return_t **host_info**(host, flavor, *info, *count). Return information about host. This is a change to the calling sequence of the current host_info call. Several machine-independent flavors are defined including the slot numbers of active processors; expansion to machine-dependent flavors to support the return of important machine-dependent configuration information is anticipated.
  - kern_return_t **host_kernel_version**(host, *version). Return the version string for the kernel executing on the given host. Experience indicates that this is the only reliable way to determine what’s actually running; version numbers are inadequate.
  - kern_return_t **host_processor_sets**(host, *processor_set_list, *processor_set_count) Obtain a list of the processor sets currently existing on the host. This is a list of the name ports, not the object ports.
- Processor operations:
  - kern_return_t **processor_info**(processor, flavor, *host, *info, *count) Return information about processor including its host. Flavor allows expansion to include important machine-dependent configuration information. Currently only one machine-independent flavor is defined (basic info). The non-privileged host port is returned.
  - kern_return_t **processor_start**(processor) Start up processor. Not all machines may support this.
  - kern_return_t **processor_exit**(processor) Exit a processor from the system. May not be supported on all machines, and the ability to restart an exited processor may be machine-dependent.
  - kern_return_t **processor_control**(processor, *data, *count) Do something to the

specified processor. Declaration and interpretation of data is machine-dependent. This supports wierd console-related operations needed by the vax 6200/6300 and 8200 among others.

Start all processors functionality can be achieved by a combination of host_processors, processor_info, and processor_start calls.

- Processor set operations:
  - kern_return_t **processor_set_default**(host, *default_set_name) Returns name port for default processor set on given host. Host privileges are not required.
  - kern_return_t **host_processor_set_priv**(host_priv, set_name, *set) Returns the object port for a processor set given the name port. The privileged host port must be used.
  - kern_return_t **processor_set_create**(host, *new_set, *new_name) Create a new processor set. Both the object and name ports are returned. The new set initially has no processors, no tasks, and no threads assigned to it. The unprivileged host port is used.
  - kern_return_t **processor_set_destroy**(set) -- Destroy the specified processor set. Any assigned processors, tasks or threads are reassigned to the default set. The default processor set cannot be destroyed.
  - kern_return_t **processor_set_info**(set_name, *host, *info, *count) -- Return information about processor set including its host (non-privileged). This information includes the number of processors currently assigned.
  - kern_return_t **processor_set_tasks**(set, *task_list, *task_count) -- Obtain list of all tasks currently assigned to the processor set. Requires the processor set's object port.
  - kern_return_t **processor_set_threads**(set, *thread_list, *thread_count) -- Obtain list of all threads currently assigned to the processor set. Requires the processor set's object port.

The last two operations can be used in combination with host_processor_sets to obtain information about all tasks and/or threads on a host. They are also of obvious utility for out-of-kernel load balancing across processor sets.

- Execution control operations:
  - kern_return_t **processor_assign**(processor, processor_set, wait) -- Assign the specified processor to the specified processor set. After assignment, the processor only executes threads assigned to the set. Any previous assignment is nullified. A wait argument of TRUE causes this call to synchronize with the specified processor ceasing to execute its current thread if that thread is not in the specified set. Using a wait argument of FALSE permits application-kernel parallelism to be exploited because the kernel uses a separate internal thread (or threads) to perform processor assignment. The master processor cannot be reassigned (to be replaced by: the default processor set must have at least one processor assigned to it at all times). This is to guarantee that internal kernel threads can be executed somewhere. Redirection of device interrupts away from processors assigned to sets other than the default set is machine-dependent. All processors take clock interrupts at all times.
  - kern_return_t **processor_get_assignment**(processor, *assigned_set_name) -- Returns the name port of the processor set to which the processor is currently assigned.
  - kern_return_t **thread_assign**(thread, processor_set) -- Assign the specified thread to the specified set. After assignment the thread only executes on processors assigned to that set. Any previous assignment is nullified. This call synchronizes with the specified thread switching away from its current processor if that processor is not in the specified set. Unix system call compatibility code may temporarily force threads to

execute on the master processor.

- kern_return_t **thread_assign_default**(thread) -- thread_assign to the default processor set. Necessary because most users cannot access the privileged object port for the default processor set.
- kern_return_t **thread_get_assignment**(thread, *processor_set) -- Obtain the name port of the processor set to which the thread is currently assigned.
- kern_return_t **task_assign**(task, processor_set, assign_threads) -- Assigns task to specified processor set. This assignment is for the purposes of obtaining the initial assignment for newly created threads. Existing threads are also assigned if assign_threads is TRUE. Same details as thread_assign.
- kern_return_t **task_assign_default**(task, assign_threads) -- task_assign to the default processor set. Necessary because most users cannot access the privileged object port for the default processor set.
- kern_return_t **task_get_assignment**(task, *processor_set) -- Obtain the name port for processor set to which the task is currently assigned. This assignment applies to any new threads created within the task, but may not apply to existing threads.

All of the assign calls fail with KERN_INVALID_ARGUMENT if their arguments (processor_set, processor, task, thread) are not on the same host. Both processor_info and processor_set_info return the (non-privileged) host as part of the information; the processor_set for a task or thread can be obtained from the corresponding get_assignment call.

host_self() is a trap. host_priv_self is also trap pending implementation of an authorization/authentication mechanisms. All other primitives 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.

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.

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 Benchmarking - A single processor set of *k* processors is sufficient to benchmark 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 Benchmarking 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.