The Mach cpu_server:
An Implementation of Processor Allocation

David L.Black

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

Version of:
14 August 1990

Abstract

The CPU-Server is a user-mode server that performs processor allocation for the Mach operating system. This document describes the server and it’s user interfaces.

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. Introduction

This document describes the cpu_server, a user-mode server that performs processor allocation for the Mach operating system, and its library interfaces. The server implements processor allocation policy; the actual mechanisms are contained in the Mach kernel (see related document). The library interfaces hide the details of server interaction from applications with simple scheduling requirements. This server is an example implementation of processor allocation policy; many other policies can be implemented using the mechanisms provided by the kernel.

Not all releases of Mach support this server. It is mainly interesting on multi-processor machines, so no major effort has been made to support it on the standard uni-processor versions of Mach. However, the new releases of Mach support it on all machines. On systems that support it, this software (MiG interface to server and library interfaces) is available in the cpu library (/usr/mach/lib/libcpu.a, -lcpu command to ld).

2. Concepts

The server performs processor allocation by assigning processors to processor sets provided by its clients. This allows the clients to use and manage the processors without giving clients complete control over them; only the server has the port capabilities required to reassign processors. Processor sets are entities exported by the Mach kernel; threads assigned to a processor set run exclusively on processors assigned to that set and vice versa (with the exception of some Unix^{tm} system calls).

The server interface is designed around a class of objects called requests. A request consists of the following components:

- A run duration,
- A sequence of <processor set, number of processor> pairs.

A request is satisfied by assigning each processor set its corresponding number of processors for the run duration specified. The server enforces internal limits on the number of processors and the maximum run time. Current limits are 15 minutes and 75% or less of the processors on the system.

3. Implementation

The server satisfies requests in a greedy fashion with strict adherence to the order in which they are received. For example, if the server has 10 processors and receives requests for 4, 7, and 2 processors, it will satisfy the request for 4 first, and then the requests for 7 and 2 together. This algorithm was chosen for its simplicity and lack of starvation; more sophisticated algorithms that make better use of the processors by satisfying requests out of order could be used.

The server implementation was based on the cthreads library and the Mach Interface Generator(MiG). Two threads are used internally; one manages the assignment of processors to requests, the other manages all interactions with clients. Clients communicate with the server via rpcs; MiG-generated interfaces hide the details of the message formats. In addition the server can optionally generate notification messages to indicate that processors have been allocated to a request and that processors are about to be removed from a request; these messages can be used for internal synchronization of applications that require multiple processors for proper execution.

4. Interface

The server's interface to its clients uses remote procedure calls to implement the following primitives:

- **cpu_request_create**(server, total_processors, run_time, *delay, *request) -- create a request for *total_processors* processors for *run_time* seconds. The request object and a delay estimate are returned. The initial server port (*server*) is obtained by looking up the name "cpu_server" with the local name service.

- **cpu_request_add**(request, processor_set, processors, *processors_left) -- Add the tuple <*processor_set*, *processors*> to the specified request. Return the number of processor remaining in the request (i.e. that can be used in future **cpu_request_add** calls).

- **cpu_request_set_notify**(request, notify_port) - Ask the server to send a notification message to *notify_port* after the processors are allocated and 1 second before removing the processors. Applications that receive these messages can use them to make sure that execution only takes place while all of the processors are allocated; the end message can be used to initiate a barrier synchronization to stop cleanly, and the start message can be used to exit the barrier.

- **cpu_request_activate**(request, options, total_time, *delay) -- Activate the request (i.e. request the server to satisfy it). A maximum delay until the requested processors will be allocated is returned. No further **cpu_request_add** calls are permitted on the request. The following options are supported:
  - <u>Destroy</u> - Destroy the processor sets when the request is completed. This is intended to support naive users by preventing a program that overruns its request from going into suspended animation; destroying its processor sets forces the program back into the default processor set where it will continue to run.

  - <u>Repeat</u> - Repeat the request for *total_time*. The time for each instance of the request is specified by the run_time argument of the **cpu_request_create** operation that created the request.

- **cpu_request_activate_task**(request, options, total_time, task, *delay) -- Identical to **cpu_request_activate**, but informs the server that the application using the processor sets is a task; this allows the server to optimize assignment and removal of processors by using **task_suspend** and **task_resume** -- processor assignment is much faster if the processor is idle.

- **cpu_request_destroy**(request) - Destroy the specified request, freeing any processors that were allocated to it. If the <u>Notify</u> <u>end</u> option applies, an end notification will be sent and the freeing of the processors will be delayed by 1 second.

- **cpu_request_status**(request, *reserved_processors, *assigned_processors, *active, *options, *time) - Find out information about a request including whether it has processors assigned to it, and how long until the assigned processors will be removed, or the maximum delay until processors will be assigned to it.

- **cpu_server_info**(server, *max_time, *max_total_time, *max_processors, *delay) -- Obtain information about the server. *max_time* is the maximum time for a single request in **cpu_request_create**. *max_total_time* is the maximum total time for a repeating request created by the <u>Repeat</u> option to **cpu_request_activate** or **cpu_request_activate_task**. *max_processors* is the maximum number of processors for any request. *delay* is the current maximum delay until a new request can be satisfied. The **cpuinfo** program uses this call to determine if the cpu_server is available and what its current situation is.

5. Library Interfaces

The above server interface along with the kernel interface will be used directly by programs that require explicit control over which threads are executing on which processors at which time. For applications with less stringent processor allocation requirements, simple library interfaces which hide all of the internal details of interaction with the kernel and server may be appropriate. Four interfaces have been developed, the allocate, task, hook, and task-hook interfaces.

The allocate interface supports a single allocation of a pool of processors. It exports the following two calls:

- **allocate_processors**(num_processors, time, interactive) - Allocate *num_processors* processors for the specified time. If the time is larger than the server's maximum slice time, then a repeating request is automatically submitted. If *interactive* is TRUE, errors generate printfs, and the user is asked whether the server's maximum delay is acceptable; no allocation is performed if the answer is no.

- **deallocate_processors**() - Free the processors allocated by **allocate_processors**. This must be called by a thread in the same task as the thread that did the allocation.

allocate_processors does not return until the allocation of processors has started; it performs a **task_assign** internally so that the initial thread and all threads and tasks subsequently created share the allocated processors. If a program overruns its time allocation, it will continue to run, but without dedicated processors.

The task interface is identical to the allocate interface, but is restricted to applications consisting of a single task so that the server can exploit efficiencies available in this case (suspending the task before removing processors). The server will perform its own **task_suspend** and **task_resume** calls on the task in this case. The task interface exports the following two calls:

- **task_allocate_processors**(num_processors, num_seconds, interactive) - Identical to **allocate_processors**, but also promises the server that all or the threads using the processors will be in the current task.

- **task_deallocate_processors**() - Deallocate the processors allocated by **task_allocate_processors**.

The hook interface supports allocation of a pool of processors, with user scheduling hooks. It exports the following two calls:

- **allocate_processors_with_hooks**(num_processors, num_seconds, start_hook, end_hook, interactive) - Allocate the specified number of processors for the specified number of seconds. If the number of seconds is greater than the server's maximum slice time, a repeating request is automatically submitted. *start_hook* is called each time after the processors are allocated. Similarly, *end_hook* is called approximately 1 second before any processor deallocation. *interactive* functions identically to the previous interfaces.

- **deallocate_processors_with_hooks**() - Free the processors allocated by **allocate_processors_with_hooks**. The *end_hook* will not be called, but users should be aware of the race between this deallocate call and the server ending a time slice.

A thread must be dedicated to the **allocate_processors_with_hooks** call; i.e. the calling thread does not return until the allocation request is finished or terminated. This dedicated thread is assigned to the allocated processors (this can be reversed by performing a **thread_assign_default** or **thread_assign** as part of the first invocation of *start_hook*. All threads within its task and subsequently created tasks are also assigned (c.f. **task_assign**). Both *start_hook* and *end_hook* must return.

Finally, there is the task-hook interface, which combines the functionality of the hook interface with the server optimization of the task interface; the calls in this interface are the hook interface calls with task_ prefixed. Users should note that the interfaces are independent; processors must be deallocated with the deallocate call from the same interface that was used to allocate them.

6. Extensions

The cpu_server could be extended in a number of ways:

- Time Sensitivity - Allow more of the machine to be allocated during off peak periods.
- Request Size Sensitivity - Use a declining priority system instead of absolute ordering to speed throughput. This is a standard technique from schedulers for physical memory machines; other similar techniques are also applicable (the current processor allocation problem for multiprocessors is essentially similar to the memory allocation problem for schedulers for physical memory machines).
- User Sensitivity - Reserve some portion of the machine for certain users based on administrative decisions (e.g. they paid for some portion of the machine, so they should always be able to get at least that portion).
- Non-Uniform Topologies - For NUMA (Non Uniform Memory Access), all topology knowledge and policy is located in the server. Techniques such as first-fit and best-fit may be applicable to the processor allocation server for such a machine (e.g. the clustered Gigamax machine being built by Encore).

In addition the server and the kernel interface (thread assignment) could be used directly in a language runtime that is sophisticated enough to control and allocate processors to its threads.

Table of Contents

1. Introduction	1
2. Concepts	1
3. Implementation	1
4. Interface	2
5. Library Interfaces	3
6. Extensions	4