OSF Mach Kernel Interface Changes

by Keith Loepere

Summary of the new features and changes in OSF Mach

The process of making the Mach microkernel suitable to support industrial quality operating system environments, real-time processing, highly trusted environments and in general, to achieve performance, has led the OSF Research Institute to propose a substantial set of additions, changes and even deletions to the OSF Mach kernel specification. This paper summarizes these features and changes. Full details can be found in the OSF Mach *Kernel Principles* and *Kernel Interfaces* volumes.

IPC Performance

Problem

- Transferring data in-line in a message required four data copies — inefficient except for fairly small data
- Variable-sized data (effectively) requires the use of out-of-line data transmission
- Transferring data out-of-line was defined as a virtual copy which is inefficient except for very large data
- Out-of-line transmission was defined as whole page transfers requiring sender and receiver to agree on data alignment

- Received message overwrote send message buffer requiring construction of fresh message for every send

Solution

- Add multiple out-of-line copy mechanisms
- Redefine out-of-line transmission to permit arbitrary data sizes and arbitrary sender and receiver alignment
- Allow the specification of the target address for out-of-line data
- Allow separate send / receive buffers

Discussion

The IPC service is enhanced with new data handling options. Because of the number of (optional) arguments to **mach_msg**, there are now two interfaces:

- **mach_msg** (with just the original basic arguments)
- **mach_msg_overwrite** (which specifies separate send and receive buffers and permits out-of-line scatter)

An out-of-line region can now be sent as either a physical or virtual copy. The sender can allow the kernel to make a decision on the method of transmission (and the associated cost) or the sender can directly choose. A physical copy is provided for improved performance for small to moderate sized areas. (A physical copy also guarantees deterministic access by sender and receiver to their data.) The sending of multiple physical copies can be thought of as a form of gather operation. Only the specified data (not the full pages containing the data) is effectively transmitted.

The receiver of a message containing out-of-line regions now has two reception options. The existing dynamic allocation option causes each region to be allocated fresh memory. The new overwrite option allows the receiver to specify, for each received region, either that the region should be allocated fresh memory or that the received out-of-line memory should be written over a specified range of memory. This option allows a form of scatter (the dispersal of the out-of-line regions). It also allows reception of data directly into a desired address, with no subsequent **vm_deallocate** necessary to free allocated space.

It is possible to have separate send and receive message buffers with **mach_msg_overwrite**.

IPC Semantics

Un-Typed Data Formats

Problem

- Kernel required "typing" for all message data, limiting the allowed structure of data

Solution

- Remove type descriptors from user data

Discussion

The message format is changed such that, other than kernel managed data (port rights and out-of-line addresses), data is not typed, thereby simplifying the message format, reducing processing time and permitting language specific marshalling and format conversion strategies, including future extension of supported data types. The **mach_msg_type** and **mach_msg_type_long** structures are obsolete.

A Mach message consists of a fixed size message header followed by a message body containing data and port descriptors and zero or more data bytes.

If the MACH_MSGH_BITS_COMPLEX flag in the *msgh_bits* field is not set, then the message is a simple one described by **mach_msg_header**. In this case, the header is immediately followed by un-typed data.

If the complex flag is set, then this is a "complex" message consisting of a **mach_msg_header** structure followed by a **mach_msg_body** structure, which permits the description of out-of-line memory regions and additional port rights. The **mach_msg_body** structure contains the number of descriptors and an array of descriptors specifying the disposition (processing) to be performed for the out-of-line memory regions and additional port rights.

Following the header (and any kernel processed body), at natural alignment, given the alignment of the preceding data, can be additional (un-typed) data, up to the size of the message (*msgh_size*). This extra data typically carries information used to decode the data stream and out-of-line data.

The IPC service enforces no particularly encoding of the data values in the data stream (or in out-of-line regions). However, MIG uses the DCE NDR encoding.

Implicit Message Attributes

Problem

- Attributes of the sending or transmission of a message were needed (e.g., sender identity, time constraint, message causality)

Solution

- Extensible message trailer contains kernel established message attributes

Discussion

As part of the act of sending/transmitting a message, the kernel, in the form of the IPC service, appends implicit message attributes to the message. These attributes are added in the form of a message trailer. The caller of **mach_msg** can select the trailer form desired, permitting future changes in the trailer while allowing old MIG stubs to receive the old trailer. There is no interface that allows the sender of a message to add such attributes; these attributes are added purely by the kernel as part of the sending/transmission of the message.

Currently, the attributes present in this trailer are the message sender's identity and the message sequence number.

No-More-Senders Notifications

Problem

- No-more-sender notification requests move when a receive right moves, but does not provide the information necessary to resume no-more-sender processing.
- No-more-sender notification message does not name the port, complicating notification processing.

Solution

- When a receive right is moved, it now cancels any pending no-more-senders notification. Since the notification is associated with the binding of a receive right to a port name space, the notification message now contains the receive right's port name.

Miscellaneous

The **norma_port_location_hint** call now works given a port name instead of a port right.

VM Performance

Paging Behavior

Problem

- External memory management governs memory object semantics, not physical page management
- Kernel owns page replacement policies

Solution

- Allow client of a memory object to specify paging behavior to bias page replacement policy
- Allow manager of a memory object to specify preferred paging transfer size

Discussion

A task can specify its expected page reference pattern for a region with **vm_behavior_set**. The allowed values for the behavior are:

- VM_BEHAVIOR_DEFAULT—The system's default behavior.
- VM_BEHAVIOR_RANDOM—No particular order expected.
- VM_BEHAVIOR_SEQUENTIAL—Forward sequential order.
- VM_BEHAVIOR_RSEQNTL—Reverse sequential order.

These values bias read-ahead strategies.

A memory manager can specify a "cluster" size (in the **memory_object_perf_info** attribute structure supplied at memory object initialization)—the preferred number of pages to be paged out at a time.

The default cluster size for temporary memory objects is set with a new parameter to **vm_set_default_memory_manager**.

Clean-in-Place

Problem

- "sync" system call support required making a copy of resident pages to be sent to the manager
- A page-out of a single page wanted to include pages in its page cluster, but these extra pages required a copy

Solution

- Allow suitably privileged memory managers direct access to the resident pages without making a copy
- Require such managers to inform the kernel when they release this access

Discussion

The kernel is confronted with several problems when returning (evicting, cleaning or flushing) pages to a memory manager.

- The kernel must be prepared to evict these pages to temporary backing storage if the manager does not remove them itself in a reasonable time.
- The kernel makes eviction decisions concerning single pages. For efficiency, though, the kernel would like to return (but not evict) multiple pages to a manager (even if the others would not otherwise be considered at this time) so that the manager can update multiple pages in its backing storage at once. These additional pages want to remain accessible to their mapping tasks.
- In principle, the manager can modify the pages returned to it (they appear as normal out-of-line memory) and so the kernel must be prepared to make writable copies of the pages.

The kernel does its best to optimize the page return mechanisms. The greatest performance is achieved when the manager will not modify the pages and can be trusted to dispose of them. Special

high performance mechanisms (internally called *clean-in-place*) are used by the kernel if the following conditions are true:

- The memory manager is privileged with respect to memory resource consumption and asserts this privilege by sending its initialization reply message (**memory_object_establish_priv**) to the host control port.
- The memory manager is on the same host (node) as the memory cache object.
- The memory manager does not modify the pages.

The use of these special mechanisms affects the semantics of the data return in that the manager must explicitly indicate to the kernel when it has disposed of the pages. The manager does this by sending a **memory_object_data_return_completed** message. The pages of memory will not be freed until the kernel receives this message. As a result, no subsequent data request message will be seen by the manager until it disposes of the returned pages.

VM Semantics

Virtual Copy Consistency

Problem

- If the copy-on-write optimization was in effect and the consistency of the memory object was being controlled by the memory manager, virtual copies would become inconsistent

Solution

- Provide a way for memory managers to inform the kernel of this impending inconsistency so the kernel would bypass the virtual copy optimization by forcibly copying the data

Discussion

There is a new *should_copy* option to **memory_object_lock_request**. This option can be used by memory managers using the MEMORY_-OBJECT_COPY_DELAY strategy when they are going to perform some action that would invalidate the kernel's memory cache for the object. This option causes any delayed copy of the specified range of

the memory object to be forcibly copied at that time. (Refer to the *Server Writer's Guide* for an explanation of this situation.)

The copy call strategy was deleted, thereby deleting **memory_object_copy**.

Object Cache Termination

Problem

- Changing a memory object to no longer be cached in the kernel's object cache may terminate the object. This fact is used by file systems to terminate un-referenced file objects. The memory object termination and the object change completion messages must be properly related so as to detect the presence or absence of this case.

Solution

- Change **memory_object_change_attributes** so that the completion message (**memory_object_change_completed**) is omitted if the change terminated the object. In this way, **memory_object_terminate** is always the last kernel message pertaining to an object. Also, the change completed message can now include the memory cache control port, since it is always sent only if the object is valid.

Miscellaneous

The MEMORY_OBJECT_COPY_TEMPORARY strategy was deleted in favor of an explicit temporary attribute.

Operating System Personality Support

Memory Reservation

Problem

- The kernel randomly allocates portions of the user's address space, conflicting with its intended use

Solution

- Allow the user to reserve portions of the address space

Discussion

A reserved memory region (**vm_reserve**) is one that prevents random allocation (including out-of-line memory reception) within it. A new allocation option for **vm_allocate** and **vm_map** request allocation explicitly within reserved memory. The allowed values for the placement are:

- VM_ALLOCATE_ANYWHERE—Allocate in the next unused, non-reserved space.
- VM_ALLOCATE_FIXED—Allocate at the given address, unless it is reserved.
- VM_ALLOCATE_RESERVED—Allocate at the given address even if the space is reserved.

The only operations supported against un-allocated reserved memory are the setting of inheritance and allocation within. Being reserved is orthogonal to all other memory properties. For example, if allocation is forced within a reserved region, the region is still reserved; de-allocation of the region will keep it reserved. The region is made no longer reserved with **vm_unreserve**.

Backing Storage Synchronization

Problem

- The "msync" system call required identifying which pages in a memory range need to be updated to backing storage

Solution

- Add a new VM interface that causes the kernel to identify and supply these pages

Discussion

A client of a memory object often desires to know that changes made to the memory object are reflected on backing storage. The client makes such a request by executing **vm_msync** against a memory region. The basic response of the kernel is to clean or to flush the set of pages (depending on a flag given by the client to **vm_msync** and on the value of the memory object's *invalidate* attribute) to the manager and to engage in a synchronization dialog with the manager.

After cleaning or flushing the range, the kernel sends **memory_object_synchronize** specifying the memory object range as well as the client's flags. One of these flags is an *asynchronous* flag. If set, the client is only interested in having the pages queued to backing storage; if not set, the client is interested in the pages actually being on backing storage. When the manager has performed the specified processing, it responds with **memory_object_synchronize_completed** specifying the disposed range. The client returns from its **vm_msync** call at that time. The range specified to **memory_object_synchronize_completed** must match that of the corresponding **memory_object_synchronize** message. Although the kernel may have multiple synchronize requests outstanding at a time, it will never have overlapping requests outstanding.

Re-Mapping Memory

Problem

- The operating system server makes repeated access to certain parts of its clients' address spaces

Solution

- Allow the server to create a mapped window into the client's address space to allow direct memory access

Discussion

The new call **vm_remap** allows a target task to map the objects in a range of a source task's memory. The effect is as if the memory object representatives governing the memory range in the source task were used for **vm_map** calls in the target task. The two tasks must be on the same host (or NORMA domain) for this to work.

Thread Abort

Problem

- **thread_abort**, used to break a thread out of message waits, also aborted various recoverable and not-so-recoverable operations

Solution

- Add **thread_abort_safely** which returns an error instead of aborting non-restartable operations

Exception Processing Performance

Problem

- Requests for service of the operating system server involve system call exception messages
- Exception message carried the thread and task ports, making them expensive messages
- Additional kernel operations were required to find and adjust the target thread state

Solution

- Allow the exception message to optionally exclude these ports (used when the exception port itself can identify the excepting thread)
- Allow the exception message to contain variable thread state and to allow the exception reply to change this state

Discussion

The exception mechanism is enhanced with exception-type-specific ports. This addition allows classes of exceptions (such as those resulting from *syscall* instructions) to be easily captured. The new exception types EXC_SYSCALL and EXC_MACH_SYSCALL are added. EXC_SYSCALL is raised when a (non-Mach) system call instruction is executed; EXC_MACH_SYSCALL is raised when a Mach system call is executed.

On a per-exception type basis, the format of the exception message can be set to provide (and return) various flavors of thread state, avoiding the **thread_get_state** and **thread_set_state** calls, and also possibly saving the passage of extra rights. The old *code* and *subcode* exception message parameters have been changed into an (implementation dependent) structure. Exception messages are now of the following types chosen when the exception port is set:

- **catch_exception_raise** — provides the task and thread ports
- **catch_exception_raise_state** — provides and returns selected thread state
- **catch_exception_raise_state_identity** — provides the task and thread ports as well as providing and returning selected thread state

With the change to type specific exception ports, exception ports are no longer manipulated with the thread (or task) “special” port interfaces. The THREAD_EXCEPTION_PORT option to **thread_set_spe-**

cial_port and **thread_get_special_port**, as well as the TASK_EXCEPTION_PORT option to **task_set_special_port** and **task_get_special_port**, have been deleted. The exception ports are manipulated as follows:

- **thread_set_exception_ports** — sets an exception port, behavior and thread state flavor for a selected set of exception types
- **thread_get_exception_ports** — returns the exception port, behavior and thread state flavor for a selected set of exception types
- **thread_swap_exception_ports** — atomically sets and returns a selected set of exception ports, behaviors and thread state flavors

There is a corresponding set of task exception calls (**task_get_exception_ports**, **task_set_exception_ports** and **task_swap_exception_ports**).

The exception ports are searched in this order:

- thread exception-type specific port
- task exception-specific port

The kernel will try each in turn until it finds one that returns a success error code. If none does, the thread taking the exception is terminated.

Exception Processing Semantics

Problem

- It is possible for another thread to abort a thread out of (syscall) exception processing
- Access to exception ports allows threads to spoof the server as to which threads appear to be executing system calls

Solution

- Make **thread_abort_safely** not abort exception sequences
- Permit a server to mark an exception port as “protected”
- Prohibit **thread_abort** of a protected exception sequence
- Prohibit fetching of protected exception ports

Discussion

As otherwise specified, it is possible for a thread to access its exception ports and to be able to receive and generate exception messages which can be useful for programs such as debuggers and other intermediaries. This ability to access the exception mechanism can cause problems for exception servers, though, that need to carefully control exception versus normal processing or processing signals or the like. A server can optionally request, via the exception behavior parameter when setting exception ports, that the exception mechanism for that class be "protected". If protected, attempts to get the exception port fail, and attempts to abort a thread within exception processing fail.

profil System Call Support

Problem

- Support for the "profil" system call

Solution

- Provide messages of PC samples upon request

Discussion

The **task_sample** and **thread_sample** calls request that the kernel sample the program counter value for the thread (or all threads in a task) at clock interrupts and to send these sampled values in messages (sporadically) to a specified port. The sample messages so produced have a defined message (**receive_samples**) and MIG de-multiplexing routine (**prof_server**).

Interposition

Problem

- Not all kernel interactions permit interposition.

Solution

- Provide a way to gain control over Mach system calls
- Provide a way to disable or interpose on the remaining system call traps

Discussion

The new exception type EXC_MACH_SYSCALL is raised when a Mach system call is executed. An exception port value of the host name port indicates that these system calls should be interpreted by the microkernel in the usual way; other values permit an exception handler to gain control.

The host self port (**mach_host_self**) is now defined to return the new (inherited) task special host-self port so that tasks may interpose on this port (or disable its use).

mach_reply_port can be disabled by setting the task-self port to null.

Miscellaneous

The composite sequence: **thread_create**, **thread_set_state**, **thread_resume** is provided by **thread_create_running**.

Security

Sender's Identity

Problem

- Client tasks could arbitrarily distribute port rights thereby excessively propagating the access implied by these rights

Solution

- Tag tasks with an opaque security identity token
- Provide a privileged way to set these tokens
- Supply the identity (security token) of the sender of a message to the receiver

Discussion

Each task is now labeled with a security identity (an opaque token). The kernel does not interpret these tokens. The security token for a task is inherited from its parent but can be set explicitly with the privileged **task_create_security_token** or changed with the privileged **task_set_security_token** call. This call requires the

new (privileged) host security port, supplied to the bootstrap task. **mach_msg** allows the receiver of a message to be given the security token of the sending task; the MIG *serversectoken* and *usersectoken* keywords allows access via MIG. A secure server would use this value when mediating access for a service request.

Memory Object Representatives

Problem

- Holder of abstract memory object port could spoof memory manager
- Abstract memory object port identifies the memory object but not the access modes permitted

Solution

- Create notion of memory object representative ports which imply the memory object, associated access modes and allowed user identity
- Distribute memory object representative ports to clients, not abstract memory object ports
- New memory object initialization sequence that authenticates the kernel to the memory manager (and vice-versa)

Discussion

Because of the integrity problems and inability to specify per-client access modes associated with abstract memory object ports, the memory object initialization sequence has been changed to use *memory object representatives*, ports that indirectly name both the associated (actual) abstract memory object ports as well as the allowed memory access modes.

The kernel's **memory_object_init** message is replaced by **memory_object_notify**. The manager now responds with **memory_object_establish** (the replacement for **memory_object_ready**) or **memory_object_representative**, which supply the representative to abstract object association.

The **vm_map** call causes the kernel to send a **memory_object_notify** message including a send right to a memory cache control port. The memory manager sends its reply message to the host name port supplied in the notify message. If the object has not been initialized, the manager responds with **memory_object_establish** specifying the ab-

stract memory object port, memory cache control port and the object attributes as well as the accesses allowed by the memory object representative. It is permissible that the memory object representative port be specified as the abstract memory object port, although this introduces a potentially vulnerability for the manager. If the kernel was not expecting the **memory_object_establish** message (a mischievous client forged the original **memory_object_notify** message), the kernel responds with **memory_object_rejected**. This informs the manager that the port it thinks is a memory cache control port really isn't.

The above sequence concerns the initialization of a memory object not yet known to the kernel. It is invoked whenever the kernel is presented (via **vm_map**) with a memory object representative port it does not know. Note that the memory object underlying the representative may well be known to the kernel; the kernel just doesn't know the association and so the kernel must assume that this representative implies the initialization of a new memory object. If the memory object is initialized, for subsequent **memory_object_notify** messages (from the same kernel) the memory manager responds with **memory_object_representative**. This call only supplies the abstract memory object port and allowed access modes for this representative

After this sequence is complete, the memory manager must be ready to respond to requests on behalf of this memory object.

Tasks using different representatives to access a memory object share the pages of the memory object as if they used the same representative, although the different tasks may have differing allowed access modes. The destruction of a representative revokes access via that representative to the underlying memory object but has no affect on tasks using different representatives.

Each representative can also be established (via the **memory_object_establish** or **memory_object_representative** message) to indicate that it applies to only a single security identity. Attempts by tasks to map such a tagged representative with a different security token will fail.

The memory cache name port is no longer sent in the initialization message (**memory_object_notify** or **memory_object_create**) nor in the termination message (**memory_object_terminate**). This is an optimization to avoid sending a right most managers do not want and also to avoid generating extra name ports that may be de-

stroyed (for representatives). The name port can be found with **memory_object_get_name**.

The port returned by **device_map** is a memory object representative port representing the desired accesses permitted by the device driver to the memory. Likewise, the port returned by **clock_map_time** (see below) is a memory object representative representing read access.

Device Access

Problem

- Given a single device port for a device no matter how many different openings were in effect, the holder of the device port was not restricted by the opening modes as to the use of the device

Solution

- Create unique device ports for each independent open of a device
- Each device port names a device, the associated access modes and allowed user identity

Information Flow through Kernel Objects

Problem

- Some calls directed at an object's name port return system global information

Solution

- Separate unprivileged information from privileged information
- Add new "statistics" calls, requiring the object privilege port, for the access of this privileged information

Discussion

Host privilege is now required to return the list of all processor sets (**host_processor_sets**).

The HOST_LOAD_INFO variant of **host_info** has been moved. This information can now be obtained by the HOST_LOAD_INFO variant of the complementary **host_statistics** call.

The virtual memory statistics (from the now obsolete **vm_statistics**) are also privileged and returned instead by the HOST_VM_INFO statistics flavor of **host_statistics**. The only piece of information that

vm_statistics used to provide that was of general interest is the page size. For this there is a new interface, **host_page_size**.

The load information (the *task_count*, *thread_count*, *load_average* and *mach_factor* fields of *processor_set_basic_info_t*) is removed from the PROCESSOR_SET_BASIC_INFO variant of **processor_set_info**, leaving just the processor count. The new complement to **processor_set_info**, **processor_set_statistics**, with the PROCESSOR_SET_LOAD_INFO flavor, returns these values.

Miscellaneous

All calls that take a privilege port just for the sake of proving privilege send their message to the object being manipulated, not the privilege port. This affects the **thread_wire** call.

Resource Management

Kernel Exhaustion

Problem

- Client tasks could consume arbitrary kernel memory
- Client tasks could consume arbitrary system paging space
- Client tasks could consume arbitrary resources on other nodes

Solution

- Create “ledger” kernel objects that count and limit kernel resource consumption (kernel memory and system backing space)
- Require a ledger when creating kernel objects
- Account for resources consumed in messages as they move
- Provide a way for a ledger to locate (via interaction with a supplied resource server) ledgers on other nodes transparently
- Charge wired memory against the task’s ledger

Discussion

Some Mach kernel resources are “created” at system initialization (such as clocks and devices) and therefore need no resource ac-

counting. The ability to create kernel objects (and therefore consume kernel resources) dynamically can lead to the ability to consume excess resources thereby leading to denial of system service. Resource *ledgers* provide the mechanism to limit such excess consumption.

A resource ledger contains a limit and the current usage count for a given resource. Each ledger limits a particular type of kernel resource:

- Amount of wired kernel memory (data structures, wired pages)
- Amount of (temporary) pageable memory (memory backed by the default memory manager).

With the exception of tasks and threads, every dynamically created, first-class (port named) kernel object consumes wired kernel memory and "draws" that kernel memory from a ledger specified when the object was created. A ledger reference is required in the **memory_object_establish**, **processor_set_create** and **device_open** calls. (**clock_alarm** also needs a ledger reference for the alarm.) These objects (memory cache objects, processor sets and devices) only consume kernel wired memory; they do not consume default memory manager space (any out-of-line memory sent to them is either wired in the process or immediately used and discarded).

In contrast, tasks (and their logical contents) consume all of the various types of kernel resources (kernel wired memory and default memory manager space). When a task is created (**task_create** or the NORMA equivalents), a set of ledgers, one for each pool of kernel managed resources, must be supplied. (If they are not, the corresponding ledger from the parent task is used.) The resources consumed by threads are considered to be associated with their containing task.

Non-first-class objects are drawn from the object that "owns" them. For example, a right in a task's port name space is counted against the task. A port (including the queued messages) is counted against the task holding the port's receive right. A receive right in transit is counted against the transferring port (which means the task holding that port's receive right).

When a first-class object is created, the creation message must carry a (send) right to one or more ledgers. A ledger represents resource consumption on a single kernel. If the supplied right indeed names a ledger on the kernel creating the object, that ledger is used as the source of the resources. However, if the right is anything other than a local ledger, the kernel sends a **ledger_get_remote** message to that right, which includes the kernel's host name port. If the response contains a local ledger port, the kernel uses it, otherwise it sends **ledger_get_remote** to that right.

This mechanism supports distributed resource ledgers in the following way:

- With **ledger_set_remote**, a ledger is assigned a "remote ledger service port".
- This ledger is used as the ledger for a create operation. If the ledger is local to the target kernel, all is fine.
- For a non-local creation, the target kernel sees that the supplied ledger is not a local ledger. The kernel sends a **ledger_get_remote** message to it, including the host name.
- The (remote) ledger receives this message, ignores the host name and returns the remote ledger service port.
- Assuming that the remote ledger service port is not a local ledger, the kernel sends a **ledger_get_remote** message to this service port.
- A server holding the receive right on the remote ledger service port uses the identity of the service port as well as the host name of the target kernel to locate or create a suitable ledger on that kernel.
- The port for a ledger on the target kernel is sent to that kernel and used.

When a ledger is destroyed (**ledger_terminate**), all objects drawn from it are destroyed.

The current limit and balance for a ledger can be found with **ledger_read**.

A ledger is created from a parent ledger with **ledger_create**. This creates a sub-ordinate ledger (which is drawn from the parent). At creation time, some of the parent's resources (some of its limit representing its ability to consume resources) can be transferred to the child. Resource limits can be transferred back and forth between a parent ledger and a child with **ledger_transfer**. Note that the ability to consume resources is transferred — a check to determine if an allocation will exceed a ledger examines only the ledger from which the object is directly drawn, not any of that ledger's parents.

The ledger ports from which a task draws its resources are task special ports. Changing these special port values, though, does not change the assignment of from which ledger the task draws its resources.

The bootstrap message used to provide the host privilege and device master ports to the bootstrap task now includes the "root" wired and

paged ledger ports. The root ledger ports for a node are NORMA special ports.

vm_wire no longer needs the host privilege port to regulate this call; wired memory is simply charged against the task's wired kernel memory limit.

Server Exhaustion

Problem

- Client tasks could flood servers with data

Solution

- Allow limits to be set on data passed in a message on a port (**mach_port_limits** attribute structure).

Real-Time

Time Services

Problem

- The kernel knew what time it was but didn't tell anyone

Solution

- Kernel clock objects that track the time and provide notification when a desired time is reached

Discussion

The kernel *clock* facility provides an explicit and well-defined interface to timeout services. The principal features of the clock facility are:

- Efficiency—The clock facility provides a basic, direct and efficient set of synchronous and asynchronous real-time alarm services.
- Granularity—The clock facility provides for dynamically changing the alarm resolution of the system. For example, the normal alarm resolution of the Mach kernel is 10-milliseconds since this is the minimum interval at which the kernel requires clock service. If a

given real-time application, however, requires alarms delivered to 1-millisecond accuracy, the clock facility can be utilized to meet that resolution.

- Relative/Absolute Alarms—The clock facility supports alarm services based on both relative (time interval measured from "now") and absolute time values.
- Time-stamps—The clock facility provides time-stamps on all alarms so that client applications can determine if requested time delays were achieved or not.

The clock facility implements (one or more) clock objects. Each host (or node) provides at least one clock, the REALTIME clock, which measures with (at least) moderate resolution the time since system initialization. This clock must provide alarm services. There may be other clocks, such as the standard defined CALENDAR clock which has lower resolution and whose time measure is not affected by system power failures or service outages, and the standard defined HIGHRES clock which, if present, allows for high-resolution alarm services. The number and nature of the clocks exported by the kernel is typically a function of the clock/-timer devices provided by the hardware, but any given implementation may provide other clocks abstracted from the hardware clocks.

A clock provides a monotonically increasing time value expressed in nanoseconds. The precision with which a clock measures the passage of time is expressed by a set of resolution values. A clock's *minimum* resolution expresses the finest precision that the clock can achieve in noting the passage of time. A clock's *maximum* resolution represents the coarsest precision at which the clock can be operated. A clock's *current* resolution, which must be a multiple of the clock's minimum resolution and less than or equal to the clock's maximum resolution, expresses the precision with which alarms and sleeps will be serviced by the clock.

The REALTIME clock provides a time reference whose initial value is set to zero by the kernel at system boot time. It is the responsibility of user-level code to translate the time tracked by the REALTIME clock into some notion of universal time, or to provide a user view of externally synchronized time. The CALENDAR clock, in contrast, provides a time value that is controlled exclusively by user-level code and that survives power or system outages.

Each clock provided by the clock facility is named by both a name port obtained by **host_get_clock_service**, which supports general clock functions, and a control port obtained by **host_get_clock_control** (which requires the host control port), which is used only to set the time or resolution of the clock.

Each clock maintains a time which whose initial value is implementation dependent. This time can be read with **clock_get_time**. The resultant **tvalspec** value consists of the current second and the nanosecond within the current second.

The implementation of a clock may provide a memory object whose contents is a mapped version of the clock time (**clock_map_time**). The form of this mapped clock (**mapped_tvalspec**) consists of a **tvalspec** value and a check value used to synchronize with the kernel's setting of the value. The REALTIME clock must provide a mapped clock.

It is assumed that the clock's time is never explicitly changed (user code is supposed to translate the implementation time into universal time and maintain any notion of synchronized clocks) but the privileged task may call **clock_set_time**.

The resolution with which alarms can be set is given by **clock_get_res** (with resolutions expressed in nanoseconds) and can be changed by **clock_set_res**.

The REALTIME clock (and, perhaps, some other implementation-defined clocks) provide sleep and wake-up services. The **clock_sleep** trap (not message) delays the calling thread until the sleep time (which can be absolute or relative according to the sleep type argument) comes. This sleep can be interrupted by **thread_abort**, which causes the call to return KERN_ABORTED. Either way, the wake time argument specifies the time the sleep was terminated.

For asynchronous time notifications, a clock also supplies the **clock_alarm** call. This call will cause the clock object to send a **clock_alarm_reply** message at the specified time (containing the alarm time). To handle this message, there is the MIG generated message demultiplexing routine **clock_reply_server**.

The **host_adjust_time**, **host_get_time** and **host_set_time** interfaces are obsolete. The *clock* pseudo-device is obsolete.

The **time_value** structure is replaced by the Posix **tvalspec**. The **mapped_time_value** structure is replaced by **mapped_tvalspec**.

Scheduling Control

Problem

- The scheduling policy interfaces supported only priority-based schemes
- Limits on scheduling priority and policy applied system-wide, thereby not permitting some tasks with greater scheduling ability than others

Solution

- New set of generic scheduling attribute interfaces
- Added task policy limits and processor set policy control

Discussion

Any given thread has associated with it a scheduling policy to use, and various parameters that influence that policy, for example, a base priority for the timeshare policy.

Any given processor set supports the scheduling of its threads according to the scheduling policies implemented for that processor set. Each thread has an associated scheduling policy which it inherits from its containing task. The (processor set) privileged call **task_set_policy** allows a task to be assigned any policy implemented by its assigned processor set. This policy can be viewed with **task_info**. Changing the policy for a task does not affect any contained threads unless an option to the set policy call is used. The (processor set) privileged call **thread_set_policy** allows a thread to assign any scheduling policy implemented by its assigned processor set. This policy can be viewed by **thread_info**. Since some of the implemented scheduling policies permit threads to overly consume processor time, a subset of the implemented policies may be enabled for random tasks/threads with **processor_set_policy_control**. The currently enabled set can be found with **processor_set_info**. A task (without privilege) may select a policy from this subset of enabled policies with **task_policy** and a thread with **thread_policy**. When a task's/thread's processor set assignment is changed, it maintains its policy unless the new processor set does not have it enabled, in which case the task's/thread's policy is set to the processor set's default policy (**processor_set_policy_control**). Disabling a policy for a processor set does not affect any tasks/threads currently in that set (unless an option to the **processor_set_policy_control** call is used).

For a given policy, a task/thread has a base set of scheduling attributes as well as limits on the values it may choose for its base attributes. (The active attributes are called the *base* set because some policies, timeshare for example, maintain a dynamic (current) set derived from the base set.)

When a task/thread is created, it inherits the scheduling limits of its parent. When a task/thread has its processor set assignment changed or changes its policy, the task/thread is assigned the limits imposed on that policy by the processor set (**processor_set_policy_control**). Changing the limits for a processor set does not change the limits for currently assigned tasks/threads unless an option to the **processor_set_policy_control** call is used. The current limits for a policy for a processor set can be found with **processor_set_info**. Under no circumstances do the base attributes for a task/thread exceed its limits.

Each task/thread has an associated set of base scheduling attributes inherited from its parent (set by **task_policy** or **task_set_policy**). Changing the attributes for a task does not affect any contained threads unless an option to the set policy call is used. The (processor set) privileged calls **task_set_policy** and **thread_set_policy** allow a task/thread to set any base it wants (and the limit as well). A task/thread may select any attributes within its limits with **task_policy** or **thread_policy**. When a task's/thread's processor set assignment is changed, it maintains its base attributes unless its limits are changed by the re-assignment, in which case they are adjusted to the limits; if the task's/thread's policy is changed (to the processor set default) by the re-assignment, the base attributes are set to the processor set's default policy base set (established by **processor_set_policy_control**). The current attributes for a task can be found with **task_info**, for a thread with **thread_info**.

The existing **thread_priority**, **thread_max_priority**, **task_priority** and **processor_set_max_priority** calls are obsolete.

Kernel Devices

The entire kernel device area is in a state of transition for a variety of reasons:

- Performance (e.g., scatter/gather lists, buffer versus copy semantics)
- Support for CDLI and the like

- Driver compatibility concerns (e.g., ioctl vs. get/set status)
- Possibility user mode device drivers

This entire area should be considered as subject to change.

In the interim:

- For performance, two new traps, **device_read_overwrite** and **device_read_request_overwrite**, allow reading data directly into a specified buffer in the caller's address space.
- **device_write** takes a de-allocate flag for the buffer.
- The device read and write calls all return an explicit device completion status code, to allow the operations to partially succeed (return a success code) and yet indicate the extent/nature of the partial failure.

New Attribute Interfaces

Problem:

- Steadily growing number of interfaces to set or get an object attribute

Solution:

- Generic set/get attribute interfaces

Discussion

For each of the existing sets of kernel objects, there are a variety of interfaces that set or get attributes for that class of object. Although many of these get or set interfaces perform special functions requiring special parameters, other interfaces simply set or get generic attributes that govern the future behavior of the object.

Many of the research efforts involving Mach are directed toward changing / controlling the behavioral aspects of certain kernel objects. These projects want a way to set and get their special attributes.

The main classes of kernel objects are given new kernel interfaces to get and set generic attributes. The intent is that a given implementation of the Mach kernel would implement a small number of attributes for each of these classes of kernel objects. It is not intended that random users be able to establish arbitrary attributes;

the set of attributes and the behavior affected by setting them is a function of the implementation of the kernel.

The kernel objects for which general attribute interfaces apply are memory objects, ports and processor sets. Some interest has been expressed for adding special attributes to tasks and hosts as well, but no specific attributes have yet been identified for those objects.

The existing get and set attribute interfaces for these classes of kernel objects are being changed to generic forms involving flavors of attributes. An attribute set is specified by the two parameters:

- An identifier (integer) indicating the flavor of attributes.
- A pointer to the attribute structure.

Ports

Port attributes include the system defined attributes (port limits) as well as implementation defined port behavior. These attributes can be manipulated with **mach_port_get_attributes** and **mach_port_set_attributes**.

The system defined flavors are MACH_PORT_LIMITS_INFO and MACH_PORT_RECEIVE_STATUS (read-only). MACH_PORT_LIMITS_INFO takes a structure containing a queue limit, out-of-line transfer limit and port right transfer limit.

The old **mach_port_get_receive_status** is superseded by the MACH_PORT_RECEIVE_STATUS variant to **mach_port_get_attributes**.

Memory Objects

The existing memory object attribute interfaces (**memory_object_change_attributes, memory_object_change_completed** and **memory_object_get_attributes**) are generalized to take a flavor of attributes.

One flavor of attributes is the set of performance attributes, MEMORY_OBJECT_PERF_INFO, which contains the cluster size. The other flavor, MEMORY_OBJECT_BEHAVIOR_INFO, contains the copy strategy and invalidate flags.

There is also a flavored version of the user-side call **vm_region** which returns a flavor of attributes (as well as the memory cache name port). The VM_REGION_INFO flavor returns all of the discrete values from the old **vm_region** call.

Processor Sets

processor_set_policy_control subsumes the functionality of the old processor set scheduling control interfaces, **processor_set_-max_priority**, **processor_set_policy_disable**, **processor_set_policy_enable**. **processor_set_info** returns this information.

Miscellaneous

A new call, **host_processor_slots**, returns the slot numbers for the configured processors. This functionality was deleted from **host_info** so that **host_info** returns only structures.

MIG Improvement / Cleanup

Some of the existing kernel interfaces had parameter sequences motivated by limitations in MIG, either then or now. Given the new improved MIG, a clean-up is in order.

Given the ability to return a single length value for calls that return multiple arrays of the same length, the redundant type count parameters are deleted from various calls. This feature is heavily used in the exception interfaces to reduce redundant count parameters.

Statistics and info calls that take or return a flavored structure now supply only the flavor and the structure. The MIG stubs know the correct length given the flavor.

device_write now accepts a de-allocate flag for disposition of the sent memory.

Note especially that kernel interfaces now take specific types for the port rights manipulated, for example, *task_port_t* instead of *mach_port_t*.

Deletions

This section summarizes features that were deleted with no equivalent replacement.

Obsolete Interfaces

The following interfaces have been deleted because they were already made obsolete by other existing interfaces:

- **mach_msg_send** → **mach_msg**
- **mach_msg_receive** → **mach_msg**
- **memory_object_data_provided** → **memory_object_data_supply**
- **memory_object_data_write** → **memory_object_data_return**
- **memory_object_set_attributes** → **memory_object_change_attributes**
- **swtch** → **thread_switch**
- **swtch_pri** → **thread_switch**
- **task_set_child_node** → **norma_task_create**
- **task_set_emulation** → **task_set_emulation_vector**
- Mach 2.5 IPC

Unused Features

Message accepted notifications (**mach_msg**) were deleted, thereby deleting **do_mach_notify_msg_accepted** and the MACH_PORT_-TYPE_MAREQUEST flag in **mach_port_type**.

The copy call strategy (**memory_object_ready**) was deleted, thereby deleting **memory_object_copy**.

The *offset* parameter has been deleted from **device_map**.

Miscellaneous

Port destroyed notifications (**mach_port_request_notification**) have been deleted, thereby deleting **do_mach_notify_port_destroyed** and the *mps_pdrequest* flag in the **mach_port_status** structure. These notifications were used only once, in the service server, in a way that was insecure. The motivating factor for deleting the notifications is that they were intended as a form of fault tolerance to handle the death of the receiver of a port. However, in the important case where this tolerance is desired, the distributed case, moving the receive right at node termination would lose any queued messages, thereby breaking the fault tolerance.

evc_wait was deleted pending a design of user-space device driver support.