Locking and Reference Counting in the Mach Kernel

David L. Black*, Avadis Tevanian, Jr.†, David B. Golub, and Michael W. Young‡§
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Coordination of independently executing threads of control within the operating system kernel is an important problem that must be addressed in the design of a multiprocessor operating system. The efficient coordination of operations is vital for avoiding performance bottlenecks without compromising system correctness. The Mach operating system achieves this coordination via carefully designed locking and reference counting techniques. This paper describes the design rationale for these techniques, and their use in the Mach operating system based on the experience of its implementors.

1 Introduction

An important problem that must be faced in the design of a multiprocessor operating system is the coordination of independently executing threads of control within the operating system kernel. Efficient and effective coordination facilities are necessary to preserve the integrity of kernel data structures without introducing performance bottlenecks. The required coordination can be subdivided into two classes, existence and operation coordination. Existence coordination ensures that a data structure exists whenever any processor could dereference a pointer to it (or otherwise access it). Operation coordination enforces restrictions on the use of a data structure to ensure that all operations return consistent results and leave the data structure in a consistent state. The simplest form of operation coordination is multiple exclusion, in which only one operation at a time is permitted. This paper examines alternatives for implementing existence and operation coordination, and describes the techniques used by the Mach kernel.

2 Coordination Techniques

The two major alternatives for implementing existence coordination are reference counting and garbage collection. Reference counting techniques maintain an accurate count of the uses of a data structure, and only permit the data structure to be destroyed when the count reaches zero. The uses that are counted include pointers to the data structure from elsewhere in the kernel, and operations in progress (such

* Current address is Open Software Foundation, Cambridge, MA.

† Current address is NeXT, Inc., Redwood City, CA.

‡ Current address is Transarc Corp., Pittsburgh, PA.

§ This Research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, ARPA Order No. 5993. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

operations often cache pointers in local variables). An incrementer of this use count is referred to as 'holding a reference'. Not all uses need be counted, as long as the count cannot reach zero while the data structure is still accessible (e.g., if two pointers to the same structure are always created and destroyed together, only one of them need be counted). Garbage collection techniques postpone evaluation of these use counts until an attempt is made to reclaim the data structure memory. A possible garbage collection technique is to stop the system, scan all of memory and reclaim those data structures for which there are no pointers. Operating system kernels cannot use this type of stop and scan technique due to its performance characteristics (the operating system stops completely during the scan), but more recently developed incremental techniques [1] are amenable to use in kernels. The Mach kernel uses a reference counting technique for existence coordination, in part because garbage collection was not viable for the C language when implementation began.

Hardware implementations of multiprocessor locking (e.g., test and set) are the basis for most operation coordination techniques. A simple test and set lock is initialized to 0, and is acquired via an atomic (hardware) test and set operation. This operation sets the lock to 1 and returns its old value; the lock has been successfully acquired if the returned value is 0. The successful acquirer subsequently releases the lock by resetting its value to 0, and unsuccessful acquirers usually spin until they are successful. Hardware engineers have implemented numerous variants of this functionality (e.g., set more than one bit [9], swap 0 and 1 for a test and clear lock [12]), but the basic concept is that of an atomic operation that sets the lock to a known state and returns its old value. The existence of caches can cause an important change in the acquisition of test and set locks because it is important that the spinning of unsuccessful acquirers not generate cache misses to avoid wasting bus or interconnection bandwidth. If the atomic test and set operation causes a cache miss (e.g., because the caches are write through), then a test and test and set sequence is substituted. This sequence loops on the test instruction until the lock is available, and only then attempts the atomic test-and-set instruction. This avoids cache misses while the lock is not available. A further refinement is to use the test and set instruction for the first attempt, resorting to test and test and set only if the first attempt fails. This assumes that most locks in a well designed system are acquired on the first attempt.

It is possible to implement operation coordination without multiprocessor locks, but such techniques are reasonable only in situations where other restrictions ensure that only a single processor can attempt to change the data structure at a time. For the common mutual exclusion case in which multiple processors may attempt to change a data structure, techniques that do not use multiprocessor locking require an independently accessible memory cell per processor. This contrasts with multiprocessor locking solutions that use a single cell (independent of the number of processors), and employ a simpler algorithm to decide which processor may proceed. The Mach kernel's operation coordination techniques are based on multiprocessor locking, with the exception of access to timer data structures in its usage timing subsystem [5].

Multiprocessor locking techniques for operation coordination vary in terms of the entities that are controlled by locks and the sizes of these entities. At one extreme is the use of a single lock protecting all or most of the kernel, restricting kernel execution to essentially one processor at a time. A related solution employs a master processor to which execution of kernel code is restricted [16]. Refinements to this approach use locks to control access to major kernel subsystems, or smaller code modules within the kernel. In all of these cases, the fact that access to code is governed by locks restricts execution of that code to at most one processor at a time, even if different data structures are involved. If large amounts of code are locked by each lock, the resulting coarse locking structure can exhibit performance bottlenecks. The alternative is to associate locks with data structures; this allows code to execute in parallel with itself, and leads naturally to a fine-grained locking structure because locks are held only for the duration of operations on associated data structures. This is the locking approach used in the Mach kernel.

3 Mach Background

Mach is a portable multiprocessor operating system developed at Carnegie Mellon University [14]. It has been ported to and used on a variety of multiprocessor platforms, including multiprocessor VAXes (784, 6000 series, and 8000 series models), the Encore Multimax, and the Sequent Symmetry. Mach is the basis for the multiprocessor support in the Open Software Foundation's OSF/1 operating system. Mach is based on a small number of fundamental abstractions implemented by a communication oriented kernel. Most kernel operations are invoked by sending messages to the kernel, permitting transparent remote invocation over networks. One of Mach's guiding implementation principles is that it should never be necessary to write kernel code that contains race conditions; the design of Mach's locking and reference counting facilities provides the necessary tools to avoid them.

The Mach kernel exports five basic abstractions; the task, thread, port, message, and memory object. These are collectively referred to as *objects* and are represented internally by data structures and routines that manipulate them. A task is an execution environment in which threads may run, and is also the basic unit of resource allocation, consisting of a paged virtual address space and access to resources (via ports). It is represented by a task data structure and an associated memory map data structure that describes the address space. A thread is a locus of control within a task. A port is a protected communication channel with exactly one receiver and one or more senders. A message is a typed collection of data objects; communication is performed by sending messages to ports. A memory object is a region of data provided by a server that can be mapped into a task. Internally, a memory object is represented by a a data structure and three associated ports. Two of these ports (the pager ports) are used for communication between the kernel and the server that implements the memory object, and the third serves as a unique identifier of the memory object.

Locks and references in the Mach kernel ensure that specified conditions of objects and/or data structures do not change for the duration of the lock or reference. Examples of these conditions include the length or contents of the object and the existence of the data structure representing it. A reference to an object guarantees that the data structure representing the object exists (i.e., it is safe to dereference a pointer to the data structure), but makes no guarantees about the existence or state of the object (alive, dead, busy, etc.). In particular, it is possible for an object to be terminated, but its data structure to remain while pointers to it exist. A lock on an object excludes certain operations (depending on both the object and the lock) while the lock is held. Any code that depends on the state of an object or its existence as an object (and not just a data structure) must hold a lock of some form.

Kernel abstractions are exported to user tasks by ports; if the abstraction is not a port, then the port data structure contains a pointer to the actual object. Operations on objects are invoked by sending messages to the corresponding ports; the object is determined from the port and the operation invoked in response to receipt of the message. Results from most kernel operations are returned to the sender in a second message; this pair of messages constitutes a remote procedure call (RPC) to the kernel. These messages are packed and unpacked by stub code generated by an interface generation tool (MiG, the Mach Interface Generator); programmers do not need to understand message formats to use kernel (or other) interfaces generated in this manner.

4 Mach Locks

Mach uses locks to prevent concurrent operations from corrupting data structures or returning inconsistent results. The most common usage is to protect against operations on separate processors, but if an operation involves blocking or otherwise suspending execution, a lock can also protect against operations on the same processor by a different thread. A fundamental distinction is made between locks that may be held for the duration of a blocking operation, and locks that do not permit blocking while they are held.

The Mach kernel uses the following locking protocols:

Simple: a spinning (non-blocking) mutual exclusion lock.

Multiple: multiple readers, single writer lock.

Sleep: allows the lock holder to suspend or wait for an event without releasing the lock.

Recursive: allows recursive acquisition and release of a lock by a single holder.

The implementation is divided into machine dependent simple locks (**Simple**) and machine independent complex locks (all other protocols). Complex locks are based on simple locks (a simple lock protects the internal state of each complex lock), so the only machine dependency is the simple lock implementation.

Machine dependent code may provide its own complex lock implementation for performance, but this is not required for completeness or correctness. Holding of a lock is always associated with a thread in the Mach kernel; a thread that acquires the lock is said to hold the lock, and is expected to perform the corresponding release operation.

Simple locks implement the **Simple** locking protocol; lock requestors actively spin to acquire the lock, and may not suspend or block without releasing the lock first[1]. The size of a simple lock (a C integer) and the interface for manipulating it are machine independent, but the implementation of the interface is machine dependent. The lock size must be large enough to hold the operand of the test-and-set or similar operation; a C integer has been sufficient on all architectures we have encountered to date. The interface to simple locks consists of an initialization routine (initialize to unlocked state), a lock routine (spin until lock acquired), an unlock routine (unlock), and an attempt lock routine (make a single attempt to acquire the lock, return success or failure). In addition, lock storage is declared by a macro that allows a storage class to be prefixed to the C storage declaration (structure containing an int). One example of the use of this prefix is to declare a lock static. The simple lock interface and declaration macro are documented in Appendix A. The implementation of simple locks usually employs a test-and-set or similar atomic operation support in hardware. Examples include the Vax bbssi (branch on bit set and set interlocked)[7] and ns32000 sbitib (set bit in byte interlocked)[13] instructions. Mach uses simple locks to protect most of its kernel data structures (e.g., task, thread, port, memory object).

Complex locks implement the **Multiple** locking protocol, and the **Sleep** and **Recursive** protocols as options to it. The **Multiple** locking protocol implements a multiple readers/single writer lock, with writers priority to avoid starvation. This means that readers may not be added to a lock held for reading in the presence of an outstanding write request, thus ensuring that the lock will be released and made available to the writer. The interface includes upgrade (read to write), downgrade (write to read) and attempt routines. Upgrades present the possibility of deadlock; this is avoided by favoring upgrades over writes, and causing upgrades to fail (releasing their read locks) in the presence of another upgrade request.

The **Sleep** option supports situations in which blocking operations will be executed while a lock is held. Examples of these operations include memory allocation (blocks if memory is not available) accessing pageable memory (may cause a page fault that could block if memory allocation is required). The **Sleep** option implements a blocking lock by blocking requestors if the lock is not immediately available, permitting lock holders to block while holding the lock. The **Sleep** option can be enabled or disabled on a dynamic basis for each lock. If a lock holder can block for any reason, the lock must have the Sleep option enabled. Most complex locks use the sleep option, including the lock on a memory map data structure.

The **Recursive** option allows a single holder to recursively acquire multiple instances of the same complex lock, allowing recursive invocation of a function that locks a particular lock during this invocation. In the absence of this option, a function calling itself on the same data structures deadlocks immediately; the second invocation tries to lock a lock held by the first invocation, but the first invocation will not release the lock until the second invocation completes. The lock must be held for writing when it is set recursive, but may be subsequently downgraded to reading; this downgrade prohibits recursive acquisitions for write and upgrades of recursive read acquisitions. If a recursive lock is held only for reading, then other read requestors may acquire it in parallel. The difference between these other requestors and the recursive holder is that the holder's requests are not blocked by a pending write or upgrade request. This permits the recursive lock holder to complete operations that require the lock (possibly increasing the recursive depth at which the lock is held in the process), so that it can drop the lock for the write or upgrade. Uses of recursive locking are rare and often indicative of locking problems elsewhere in the system.

[1] This is a design requirement; violations of this restriction cause kernel deadlocks.

5 Lock Usage

Mach's locking subsystem implements lock manipulation routines (e.g., lock, unlock), but does not control allocation of lock data structures. Instead it exports declaration macros and initialization routines to other subsystems, which are responsible for lock declaration and management. This provides a high degree of flexibility to kernel implementors in designing locking protocols and managing data structures protected by locks. The most common use of locks is to control access to dynamically allocated data structures. This is typically implemented by declaring a lock as part of the data structure and initializing it in the corresponding allocation routine. The overall locking philosophy is to lock data structures in preference to code; this allows routines to run in parallel with themselves on different processors when different data structures are involved.

Simple locks are used in a variety of ways. The simplest form of lock usage employs a single lock that locks an entire data structure. For static data structures, this lock may be declared as a separate variable. Some classes of objects have more than one lock in order to allow concurrent operations on different parts of the object (e.g., a task has two locks to allow task operations and ipc translations to occur in parallel). A final alternative is to implement a customized lock based on a data structure locked by a simple lock. One example of this is that the memory object structure contains flags that indicate whether paging ports are being created for an object to ensure that the ports are created at most once. A simple lock cannot be held during this operation, because the allocation of the port data structures may block. Instead, a boolean flag is set to indicate that the operation is in progress and a second one is set when the operation is complete. Both of these flags are set while holding a simple lock on the memory object structure, making these flags a customized lock that extends the functionality of the simple lock on that data structure.

Each kernel subsystem that uses locks must incorporate usage conventions that prevent deadlock, because the range of possible locking protocols precludes a single lock hierarchy and similar deadlock avoidance schemes. The simplest such convention is to order lock acquisitions by object type (e.g., always lock the memory map before the memory object). If two objects of the same type must be locked, the acquisitions can be ordered by address. A more complicated example can be found in the machine dependent portion of Mach's virtual memory system (pmap modules) for tree structured memory mangement units. These modules manage two classes of data structures, the physical maps (pmaps), and physical to virtual lists (pv_lists). The physical map maintains virtual to physical mappings in the format required by the MMU hardware, and the pv_lists provide an inverted list for determining the virtual mappings of physical memory. Both data structures have locks, and the pmap modules contain routines that need to acquire these locks in both orders (pmap then pv_list, and pv_list, then pmap). To resolve this conflict, a third lock (the pmap system lock) is used to arbitrate between the orders in which these locks may be acquired. In some systems this is a readers/writers lock, so that any procedure with a write lock on this lock can assume exclusive access to the pv lists. Other modules have implemented this as a custom designed lock for this situation (two exclusive classes of readers). A final alternative is to use a backout protocol when acquiring two locks in the reverse of the usual order; a single attempt is made for the second lock, with failure causing the first one to be released and reacquired later. More information on pmap modules can be found in [15].

6 Locks and Synchronization

Mach's event wait primitives are often used as part of locking protocols. An important operation in many protocols (especially those that do not use sleep locks) is releasing one or more locks to wait for an event. This operation must be atomic with respect to the operation that declares event occurrence; this avoids races in which the event occurs while the locks are being released, leaving the waiter blocked indefinitely. Mach implements this functionality by splitting the wait functionality into declaration and conditional wait components. The following routines implement Mach's event wait mechanism:

1. `assert_wait` – Declare event to be waited for.
2. `thread_block` – Context switch. Causes wait if event has not occurred.

3. `thread_wakeup` – Event based event occurrence.
4. `clear_wait` – Thread based event occurrence.

Event occurrence synchronizes with `assert_wait`; `thread_block` blocks thread execution only if the event has not occurred since the `assert_wait`. A thread that needs to release locks and wait for an event calls `assert_wait` before releasing the locks, and `thread_block` afterwards. If the event occurs in the interim, the `thread_block` is converted to a non-blocking context switch that leaves the thread runnable. The common case of releasing a single lock to wait for an event is available via the `thread_sleep` routine. The thread based occurrence routine, `clear_wait`, is provided to allow users of the event mechanism the option of tracking blocked threads instead of relying on the event mechanism to do so. Such an implementation could block threads on event zero (the null event), from which only a `clear_wait` can awaken them. Further details on these routines and the internal functioning of Mach's scheduler can be found in [4].

7 Locks and Interrupts

The interaction of locks and interrupts causes three problems:

1. The interrupt code may try to acquire a lock held by the code that was interrupted. Interrupt routines lack the thread context required to sleep, and are therefore forbidden from acquiring sleep locks.
2. An interrupt while a lock is held may cause a critical lock to be held for an excessive period of time (duration of the interrupt service routine).
3. Inconsistent interrupt protection can cause barrier synchronization at interrupt level to deadlock.

The first two problems are solved by only holding the lock in question with the offending interrupts (all in the second case) disabled. This requires that a requestor disable interrupts before acquiring the lock and re-enable them only after releasing the lock.

The barrier synchronization at interrupt level causes a more complicated problem; all involved processors must enter the interrupt service routine before any can leave. If interrupts are disabled inconsistently, this can cause a system deadlock on multiprocessors in the following fashion:

- Processor 1 has the lock with interrupts enabled. Processor 2 has disabled interrupts and is attempting to acquire the lock.
- Processor 3 initiates interrupt barrier synchronization. Processor 1 takes the interrupt, processor 2 does not.
- The system now deadlocks as follows:
  - Processor 3 is waiting for processor 2 to take its interrupt so that processor 3 can complete the synchronization.
  - Processor 2 is waiting for processor 1 to release the lock, and will not take interrupts before the lock is released.
  - Processor 1 is waiting for processor 3 to complete the synchronization.

To avoid this situation, each lock must always be acquired at the same interrupt priority level (spl0, splvm, splnet, splclock, etc.), and held at that level or higher[2]. The lock should be released at the same priority (this is a requirement for complex locks, as each operation on them locks and unlocks a simple lock protecting their contents). This notion of associating a single interrupt priority level with each lock is a good design principle. Implementors of low level modules should be aware that their modules may be called from higher level code at different interrupt priority levels (due to design considerations at

[2]The precise requirement is that locks must be acquired at the same interrupt priority level with respect to the interprocessor interrupt used for barrier synchronization. Because this priority level is machine dependent, the same interrupt level restriction ensures correctness independent of this priority level.

the higher levels). Increasing interrupt priority with increasing call depth is always safe so long as the priority is consistent for each lock. This is one of the reasons why the scheduler raises interrupt priority to its highest level (blocking all interrupts).

This solution does not cover locks held by the processor that initiates an interrupt level barrier synchronization. For the current (and we hope only) use of barrier synchronization at interrupt level, TLB shootdown, the only locks that fall into this situation are the pmap locks, and the TLB shootdown code contains special logic to deal with this situation. This logic removes a processor attempting to acquire or holding such a lock from the set of processors that must participate in the barrier synchronization. The TLB update is still posted for that processor, and an interrupt is sent to it. The processor will re-enable interrupts, and hence take this interrupt before it touches pageable memory again. For a detailed discussion of the TLB shootdown code, and why it requires barrier synchronization at interrupt level, see [2]. Barrier synchronization at interrupt level is actively discouraged because it is a costly operation.

7.1 Locking Experience

The overall experience with Mach locking has been quite positive. Basing all locking functionality on simple locks limited the machine dependencies and enhanced the portability of the system. The locking primitives have been extensively used in subsequently designed kernel subsystems (e.g., processor allocation [3]), and parallelization of UNIX compatibility code in Mach kernels [11]. The overall design and implementation have proven to be flexible and effective in adapting to the different locking requirements of the various kernel subsystems.

The read to write upgrade feature of Mach's complex locks is rarely used because a failed upgrade attempt releases the read lock. Releasing the lock in this situation is required to avoid deadlocked upgrades, but also requires recovery logic in the caller to handle failed upgrades. A simpler alternative that avoids upgrades is to initially lock for writing, and downgrade to a read lock after operations that require the write lock are complete. This downgrade cannot fail and does not require any special logic in the caller.

Recursive locks are less than fully general because they impose a restriction on routines that acquire the recursive instances of the lock. A routine that acquires a non-recursive lock can be sure that releasing that lock will allow any other routine to acquire it, and thus can release the lock as part of waiting for an event that will be caused by such a routine. The corresponding use of a recursive lock can cause deadlock because the lock is not fully released (the original acquirer still holds it), so these routines cannot be called by the holder of a recursive lock. This restriction was not fully understood at the time recursive locking was designed into the Mach kernel.

The Mach kernel routine that changes memory pageability, vm_map_pageable, was the original motivation for recursive locking and is an example of its drawbacks. When making memory non-pageable (i.e., wired or pinned), it acquires a write lock on the memory map to change the appropriate map entries, and downgrades to a recursive read lock to fault in the memory. The fault routine in turn requires a read lock on the map, and in some cases a write lock. To avoid an upgrade to a write lock, vm_map_pageable must perform any work that would otherwise necessitate a write lock in the fault routine. If one of the faults cannot be satisfied due to a physical memory shortage, the fault routine drops its lock to wait for memory. The fact that vm_map_pageable still holds a read lock can cause a deadlock if obtaining more memory requires a write lock on the same map. While these deadlocks are difficult to cause, they have been observed in practice. To eliminate them, vm_map_pageable is being rewritten to avoid the use of recursive locks. We are sufficiently dissatisfied with the utility of recursive locking that we intend to remove support for it from the Mach 3.0 microkernel [10] when this rewrite is completed. Duchamp [8] reports similar negative experiences with locking and recursive operations in a transaction processing application.

8 References

A reference is used to guarantee the existence of an object's data structure. There are several classes of references:

Direct: executing code holds a reference to the object.

Indirect: executing code holds a reference to an object which references a second object (and so on for further levels of indirection). In some cases, locks may be necessary to preserve intermediate links in this chain; these locks prevent operations that destroy one of the pointers and release the corresponding reference.

Implicit: several minor cases. Any code has an implicit reference to the current thread it is executing in. In addition, some objects are referenced by static data structures.

References are implemented by a reference count field in the corresponding data structure. Acquiring a reference increments this count; releasing a reference decrements it. The routines that increment and decrement these counts are implemented as part of each subsystem to allow flexibility in allocation and deallocation. An interesting reference counting case is that memory objects contain two independent reference counts, a reference count for the data structure and a reference count for paging operations in progress. The latter count is a hybrid of a reference and a lock because it excludes operations such as object termination that cannot be performed while paging is in progress.

Object references are used for three purposes:

Inter-object pointers: a reference on the object pointed to guarantees that the pointer is valid.

Operations in progress: a reference to the object being manipulated guarantees that the data structure representing it will not vanish during a complex operation (e.g., when it is necessary to drop all locks). In particular, a reference is necessary in order to acquire a lock in an object.

Object existence: An object is created with a single reference to itself. The creator is responsible for removing this reference when it is no longer needed.

When an object's reference count reaches zero there are no operations in progress on it, no pointers to it, and no ways to invoke new operations on it (because there are no pointers); therefore the object and its data structure can be destroyed at that time.

References can be acquired to an object in two ways:

- Invoke the create operation which returns a new object with one reference to its creator.
- Clone an existing reference by locking the object and incrementing its reference count. The existing reference ensures that the data structure does not get deallocated while the lock is being acquired. There are several ways in which references can be cloned:
  - Executing code holds a reference and clones it.
  - Executing code performs a name to object translation. This effectively clones the object reference held by the name translation data structures.
  - A reference held in a static data structure is cloned under a lock or other guarantee that the original reference will not vanish during the cloning operation.

Actually acquiring a reference requires locking the object (or the portion containing its reference count) in order to increment the reference count.

Acquiring a new reference to an object will not block, and therefore may be done while holding other locks. Releasing a reference, however, may destroy the object (if the last reference is released); this frees storage and may perform other operations that can block. Thus it may not be done while holding any non-sleep locks, nor between an assert_wait() operation and the corresponding thread_block() because the blocking operations will call assert_wait() a second time (this is fatal).

9 Deactivated Objects

An object that consists only of a data structure (on which essentially no operations can be performed) is said to be deactivated. The data structure will survive so long as there are references to it, but attempts to perform most operations on it fail because the data structure contains an indication that the object has been deactivated. Because some objects can be deactivated at any time, the following rules apply:

- If an operation depends on the object not being deactivated, this must be checked whenever the object is locked during the operation because the object can be deactivated at any time it is unlocked.
- Pointers from an object and the internal state of that object cannot, in general, be saved when unlocking and relocking the object. If an object is deactivated, pointers and the corresponding references may be removed and released. A reference is required in order to relock the object.

An operation that fails because an object has been deactivated performs whatever recovery code is required to avoid corruption of data structures and returns a failure code.

Deactivation is used for objects that are actively terminated (e.g., tasks and threads) rather than passively vanishing when the last reference to them disappears (e.g., memory maps).

10 Kernel Operations and References

A kernel operation is invoked via a msg_rpc to the kernel. For operations on objects (task, thread, etc.) this involves the following sequence of operations:

1. The request message is received. This message contains a reference to the port from which it was received.
2. The represented object is determined from the port and a reference is obtained to the object. This translation code is generated by MiG.
3. The operation executes. This usually involves acquiring and releasing at least the lock on the object and often other locks. Note that the object and its corresponding port cannot vanish due to the references acquired above.
4. The operation completes. Interface code releases the object reference. In Mach 3.0 systems this is changed slightly; a successful operation consumes (uses or releases) the object reference, so the interface code releases the reference only if the operation fails.
5. Reply message returns result. Internal destruction of original message releases the port reference.

One important case of a kernel operation is the shutdown operation. The case of most interest is an object that can be deactivated and is represented to the outside world by a port. After acquiring the reference to the object (step 2 above), shutdown is accomplished as follows:

1. Lock the object, set the "deactivated" flag, and unlock the object.
2. Lock the corresponding port, remove the object pointer and reference from the port, and unlock the port. This disables port to object translation.
3. Shutdown/destroy the object. Requires a lock.
4. Release the reference originally returned by object creation. This will cause final deletion of the object when all other references are released.

11 Conclusion

The techniques used to coordinate independently executing threads of control within an operating system kernel are an important component of a multiprocessor operating system. This paper has described the locking and reference counting techniques used to implement this coordination in the Mach operating system. The Mach implementation provides basic locking services at a level that permits great flexibility in the use, management, and customization of locks and their associated protocols. Greater implementation effort is required in comparison to a centralized locking implementation, but the flexibility gained outweighs these costs. Reference counting is closely connected with the allocation and deallocation of memory, and is therefore left to individual subsystems to manage as they so choose. The Mach experience suggests that common locking and reference counting models are useful, but that the flexibility of kernel implementors to design their own locking and reference counting protocols based on these models is very important.

References

[1] Andrew W. Appel, "Garbage Collection," Topics in Advanced Language Implementation, Peter Lee, ed, MIT Press, (1990).

[2] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron, "Translation Lookaside Buffer Consistency: A Software Approach," Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems, (April, 1989), pp. 113–122.

[3] David L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", COMPUTER, (May, 1990), pp. 35-43.

[4] David L. Black, Scheduling and Resource Management Techniques for Multiprocessors, PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, (July, 1990), Available as Technical Report CMU-CS-90-152.

[5] David L. Black, "The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing," USENIX Mach Workshop Proceedings, (October, 1990), pp. 53–72.

[6] Eric C. Cooper and Richard P. Draves, C Threads, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-88-154, (June, 1988).

[7] VAX Architecture Handbook, Digital Equipment Corporation, Maynard, MA, (1984).

[8] Dan Duchamp, "Experience with Threads and RPC in Mach", USENIX Symposium on Experiences with Distributed and Multiprocessor Systems Proceedings (SEDMS II), (March, 1991), pp. 87–104.

[9] Multimax Technical Summary, Encore Computer Corporation, Marlboro, MA, (1986).

[10] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "Unix as an Application Program," Proceedings of the Summer 1990 USENIX Conference, (June, 1990), pp. 87–95.

[11] Alan Langerman, Joseph Boykin, Susan LoVerso and Shashi Mangalat, "A Highly-Parallelized Mach-Based Vnode Filesystem," Proceedings of the Winter 1990 USENIX Conference, (January, 1990), pp. 297–312.

[12] Ruby B. Lee, "Precision Architecture", COMPUTER, (January, 1989), pp. 78–91.

[13] Series 32000 Instruction Set Reference Manual, National Semiconductor Corporation, Santa Clara, CA, (1984).

[14] Richard F. Rashid, Threads of a New System, Unix Review, (August, 1986), pp. 37–49.

[15] Avadis Tevanian, Jr., Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, (December, 1987), Available as Technical Report CMU-CS-88-106.

[16] James W. Wendorf, Operating System/Application Concurrency in Tightly-Coupled Multiprocessor Systems, PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, (August, 1987), Available as Technical Report CMU-CS-88-117.

A Simple Lock Interface

This appendix describes the kernel interface to simple locks. The declaration and storage format of simple locks is machine independent, but all routines that operate on them are machine dependent. This interface is only available within the Mach kernel, and it is not exported to user programs. Similar functionality is available in most libraries that support multithreaded applications.

A.1 Declaration and Initialization

Users of simple locks are responsible for declaring and initializing the data structures that represent them. A machine independent declaration macro and a machine dependent initialization routine are part of the interface to simple locks (e.g., the mutex functionality in the C threads library [6]).

The `decl_simple_lock_data(class,name)` declares a simple lock variable `name` with storage class `class`. A simple lock is stored in a C language `int` variable, which is part of a structure to allow the simple addition of debugging and statistics information. The storage class prefix can be used to add any valid C storage class specifier, such as `static`. A macro is used instead of a C type to allow simple locks to be defined out of uniprocessor kernels.

The `simple_lock_init(lock)` initializes a simple lock to its unlocked state. It is used only for initialization, not for unlocking a locked lock.

A.2 Locking and Unlocking

The following machine dependent routines lock and unlock simple locks. All take a single argument that is the address of the lock. The `simple_lock_addr(lock)` macro is used to obtain lock addresses so that simple locks can be defined out of uniprocessor kernels.

- `simple_lock(lock)` – Lock the lock, spinning until it is acquired.
- `simple_unlock(lock)` – Unlock the lock.
- `boolean_t simple_lock_try(lock)` – Make a single attempt to lock the lock, returning a boolean indicating success (TRUE) or failure (FALSE).

The `simple_lock_try` routine is useful for attempting to acquire a lock in situations where the unconditional acquisition of the lock could cause deadlock. Simple locks may not be held during blocking operations or context switches.

B Complex Lock Interface

This appendix describes the kernel interface to complex locks. This entire interface is machine independent. This interface is only available within the Mach kernel; it is not exported to user programs. Complex locks are implemented by a data structure which contains a simple lock to protect the state of the complex lock.

B.1 Declaration and Initialization

Complex locks must be declared and initialized by their users. The C type `lock_data_t` declares storage for a single complex lock. The C type `lock_t` is a pointer to a `lock_data_t`, and is the type of the lock arguments expected by all routines in this interface. The `lock_init(lock, can_sleep)` routine initializes a lock; `can_sleep` is a boolean indicating whether the Sleep option is desired. Locks without the sleep option cannot be held during blocking operations or context switches. Multiple locking (readers/writers) is always available, and recursive locking is enabled and disabled by separate routines.

B.2 Locking and Unlocking

The following routines lock and unlock complex locks:

- `lock_read(lock)` – Acquire the lock for reading.
- `lock_write(lock)` – Acquire the lock for writing.
- `boolean_t lock_read_to_write(lock)` – Upgrade a read lock to a write lock.
- `lock_write_to_read(lock)` – Downgrade a write lock to a read lock.
- `lock_done(lock)` – Release a lock.

The boolean returned by lock_read_to_write indicates whether the upgrade failed. If anther upgrade is pending, this upgrade fails (TRUE is returned) and the read lock is released.

A lock can be held either by a single writer or by one or more readers, thus lock_done can always determine how the lock is held and release it appropriately. All of the locking routines spin (Sleep option off) or block (Sleep option on) until the lock is acquired, with the exception of the failed upgrade case of lock_read_to_write.

B.3 Lock Attempts

The following are versions of the lock routines that make a single attempt to acquire a lock, and return a boolean indicating success or failure.

- boolean_t lock_try_read(lock) – Attempt to acquire the lock for reading.
- boolean_t lock_try_write(lock) – Attempt to acquire the lock for writing.
- boolean_t lock_try_read_to_write(lock) – Attempt to upgrade the lock from reading to writing.

The first two routines do not spin or block under any circumstances. In particular, this means that lock_try_write returns FALSE if the lock is currently held for writing. lock_try_read_to_write may

block waiting for other readers to drop the lock in the process of obtaining the upgrade. The Mach 2.5 implementation of this routine contains a bug such that it will block even if the Sleep option is disabled for the lock; this may be due in part to the fact that no Mach kernel uses this routine. lock_try_read_to_write does not drop the read lock if the upgrade would deadlock.

B.4 Lock Options

The following routines change the options for a complex lock:

- lock_sleepable(lock, can_sleep) – Enable or disable the Sleep option according to the boolean value of can_sleep.
- lock_set_recursive(lock) – Enable the Recursive option for the current (calling) thread. The lock must be held for write.
- lock_clear_recursive(lock) – Clear the Recursive option for the current (calling) thread.

lock_clear_recursive should be called by the caller of lock_set_recursive before releasing the lock.