Mach3 Locking Protocol[1]

Franco Travostino (travos@osf.org)
OSF Research Institute
1 Cambridge Center
Cambridge, MA 02139

Abstract

This paper presents an extensive study of the Mach 3.0 kernel's MP locking technology and describes, for each kernel subsystem, that subsystem's locking protocol hierarchy. We found that each subsystem has a documentable locking protocol even though there is no formal hierarchy. Consequently, there are a significant number of places where the subsystems violate their protocols.

1 General Assumptions

The existing locking architecture for MP is based on the following assumptions:

- There are two lock types: *simple* locks (often referred to as spin locks), and read/write *complex* locks. Both lock types have an interface that provides conditional locking. A complex lock may optionally cause blocking, may manage a queue of waiting threads, and may support recursive locking.

- Simple locks cannot be held across context switches.

- Memory protected by a simple lock is wired when the lock is held.

- *spls* cover the simple locks that are used to synchronize with interrupt routines.

- The memory model assumed for MP/UMA architecture allows data aligned on “natural” boundaries (e.g, a long is always allocated on a longword boundary) to be fetched atomically. In other words, the current value, or a stale value will be fetched, but the value cannot be corrupted by an interleaving write operation.

- All counters inserted in order to collect cumulative data with respect to system behavior are incremented without locks.

2 Tasks, Threads, and Scheduling

The entire locking protocol of this subsystem is based on simple locks. The locks used in this subsystem’s protocol are listed below. For each lock we provide a short description of the lock’s purpose and specify the syntax used to invoke the lock.

task_lock Protects most of the task structure. The lock is invoked as follows:

[1]This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

task_lock(*task***)**

Defined as follows:

#define task_lock simple_lock(&(task)->lock)

itk_lock Protects those fields within the task structure that are maintained by the IPC subsystem. The lock is invoked as follows:

itk_lock(*task***)**

Defined as follows:

#define itk_lock(task) simple_lock(&(task)->itk_lock_data)

thread_lock Protects most of the thread structure. The lock is invoked as follows:

thread_lock(*thread***)**

Defined as follows:

#define thread_lock(th) simple_lock(&(th)->lock)

ith_lock Protects those fields within the thread structure that are maintained by the IPC subsystem. The lock is invoked as follows:

ith_lock(*thread* **)**

Defined as follows:

#define th_lock(th) simple_lock(&(th)->ith_lock_data)

pset_lock Protects most of the processor set structure. If taken in conjunction with a thread lock, must be taken before the thread lock. Invoked as follows:

pset_lock(*pset***)**

Defined as follows:

#define pset_lock(pset) simple_lock(&(pset)->lock)

processor_lock

Protects various fields of the processor structure. It may be taken while the current thread holds the processor set's main lock (**pset_lock**). The lock is invoked as follows:

processor_lock(*pr***)**

Defined as follows:

#define processor_lock(pr)simple_lock(&(pr)->lock)

all_psets_lock Protects the **all_psets** queue, the kernel's list of all currently allocated processor sets. If taken in conjuction with a processor set lock (**pset_lock**), **all_psets_lock** must be taken first. The lock is invoked as follows:

simple_lock(&all_psets_lock)

Separate locking interface macro not defined.

action_lock Protects the **action_queue,** a queue used by the kernel to assign/shutdown processors. If taken in conjunction with a **pset->idle_lock**, the **action_lock** must be taken first. Taken in this order, these locks allow the kernel to safely check the state of a processor before transferring the processor from its processor set queue to the action queue. The lock is invoked as follows:

simple_lock(&action_lock)

Separate locking interface macro not defined.

pset->idle_lock

Protects the processor set idle queue. If taken in conjunction with a **processor_lock**, the **pset->idle_lock** must be taken after the **processor_lock**. The lock is invoked as follows:

simple_lock(&(*pset*)->idle_lock)

Separate locking interface macro not defined.

quantum_adj_lock

Protects a processor set structure's **quantum_adj_index** field. The lock is invoked as follows:

simple_lock(&(*pset*)->quantum_adj_lock)

Separate locking interface macro not defined.

runq->lock Protects the set of run queues associated with a given processor or processor set. (A run queue set contains 32 run queues, except on the MACH3/ RT version, where a run queue set contains 128 run queues.) The **runq->lock** is typically taken after several other locks have been taken because it is the lowest level data structure involved in thread scheduling. The lock is invoked as follows:

simple_lock(&(*runq*)->lock

Separate locking interface macro not defined.

wait_lock Protects a wait queue hash bucket. There is one simple lock per hash bucket. The lock is invoked as follows:

simple_lock(&*wait_lock[i]***)**

Separate locking interface macro not defined.

stack_lock Used when stacks are allocated from generic kernel VM. The lock is invoked as follows:

simple_lock(&*stack_lock***)**

Separate locking interface macro not defined.

stack_usage_lock

Related to stack_max_usage. The lock is invoked as follows:

simple_lock(&*stack_usage_lock***)**

Separate locking interface macro not defined.

swapper_lock Protects the **swapin_queue**. To queue a thread on this queue, the kernel must first lock the thread, then take the **swapper_lock**. The **wait_lock** and the **thread_lock** of the current thread may be taken while holding the **swapper_-lock** (**assert_wait(**)). The lock is invoked as follows:

simple_lock(&swapper_lock)

Separate locking interface macro not defined.

reaper_lock Governs the **reaper_queue**. The kernel can place a thread on the reaper queue without locking the thread because the thread is in a known state and cannot be transitioned to another state. While the **reaper_lock** is held, the **wait_lock** and **thread_lock** of the current thread may be taken (**assert_wait(**)). The lock is invoked as follows:

simple_lock(&reaper_lock)

Separate locking interface macro not defined.

timer_lock Protects the timein/timeout queue. There are no dependencies with other locks; it may be invoked within the interrupt stack. The lock is invoked as follows:

simple_lock(&reaper_lock)

Separate locking interface macro not defined.

2.1 The Locking Protocol with Respect to Masking Interrupts

The following set of locks cannot be invoked safely without first setting the hardware's interrupt priority via *splsched*( ) (on most platforms, synomous with *splsched*( )):

t**hread_lock** The clock handler invokes this lock when updating a thread's priority and via the thread_wakeup path**.**

runq->lock This lock is invoked in the thread_wakeup path.

wait_lock This lock is invoked in the thread_wakeup path.

processor_lock

This lock is invoked via machine-dependent implementations for performing start and stop operations on CPUs.

stack_lock Involved in interrupt stack management.

timer_lock Invoked within the clock handler.

quantum_adj_lock

Invoked within the clock handler.

2.1 Hierarchy "Rules of Thumb"

Although the subsystem does not implement a formal locking hierarchy, a number of general rules can be detected that give one the sense of a locking protocol. These are as follows:

1. New entities do not need to be locked because they cannot be referenced outside the context of the creating thread.

2. When it removes an entity from a system queue, the kernel may not need to lock the entity, because the entity also may not be referenced outside the context of the current thread. However, many of the entities implemented by this subsystem are referenced via different mechanisms, so this rule may be applied only under certain conditions. For an example, trace the algorithm used to remove threads from the run queue and begin their execution (See the comments on the thread section).

3. There are some entities that can be handled without locking when their reference counts drop to 0. (e.g. threads that are halted and are waiting to be garbage collected).

4. The subsystem does not use conditional locking statements (e.g. simple_lock_try).

5. With respect to each data structure type, there are fields that do not need to be protected because they are written by a single writer. For example, the **last_processor** field of thread A is set by thread B when that thread is switching the CPU's context to run thread A.

6. Homogeneous entities are maintained in doubly-linked lists, whose head is maintained within another entity. The following are examples:

   •Associated with each task is a list of the task's threads; the head of this queue is contained within the task structure.

   •Associated with each processor set is a list of all tasks that are assigned to the processor set., and a list of all threads that are assigned to the processor set; the heads of both queues are contained within the processor set structure.

   The kernel must be able to lock the head of the queue and then lock specific elements within the queue.

   The kernel can safely scan a given queue by locking the queue's head. If the kernel must drop the lock while scanning in order to block, the kernel must be able to account for changes in the queue's contents upon wakeup. The kernel accounts for the possibility of new entities by assuming that they are added to the end of the queue and that the queue is consistent once its head is relocked. To deal with the case that entities may have been removed from the queue, the kernel must distinguish two cases:

   •The current algorithm does not care because it is about to terminate all entities in the queue. An example of this is when a task must terminate all of its threads; a thread that is terminating all threads in a task does not care if another thread destroys a thread. However, as the terminating algorithm processes each entity in the queue, it must prevent the entity from disappearing while it is being processed. Reference counts are used for this purpose:

```
while (!queue_end(list, (queue_entry_t) thread)) {
    if(!task->active){
        ret = KERN_FAILURE;
        break;
    }
    if(thread != current_thread( )) {
        thread_reference(thread); /* reference_count++ */
        task_unlock(task);
        if(prev_thread != THREAD_NULL)
            thread_deallocate(prev_thread); /* may block */
        thread_assign(thread, new_pset);  /* may block */
        prev_thread = thread;
        task_lock(task);
    }
    thread = (thread_t) queue_next(&thread->thread_list);
}
```

   •The algorithm does care about queue entries being removed, and for some reason, the kernel wants to reach all queue entries present at some point in time. Still, the reference count mechanism helps, and the algorithm can build a list of entities that it will manipulate afterwards:

```
pset_lock(pset);
for (i = 0, thread = (thread_t) queue_first(&pset->threads);
      i <actual; i++, thread = (thread_t) queue_next(&thread->pset_threads)){
      /* take ref for convert_thread_to_port */
      thread_reference(thread);
      threads[i] = thread;
}
pset_unlock(pset);
/* manipulate threads[ ] now */
```

7. There is a notion of hierarchy among the subsystem's principal locks (multiple entries at the same level in the hierarchy indicate that there is no relation among them; a detailed description of their use will be provided):

higher position	all_psets_lock
-	pset_lock
-	task_lock
-	thread_lock, wait_lock
-	processor_lock
lower position	runq->lock, pset->idle_lock

This general hierarchy comes from the fact that the processor set structure contains the heads of two queues: a queue of tasks and a queue of threads. To scan either of these queues, the kernel locks the queue's head with **pset_lock**; once a queue entry is extracted, the kernel locks it before manipulating it.

The **all_psets_lock**, which the kernel uses to prevent processor sets from being added or removed, is the only lock that is above the **pset_lock**, and is invoked only when the kernel performs operations that cross processor set boundaries.

Below the **thread_lock** are the locks the kernel takes to dispatch runnable threads. The dispatching operation either places the thread on a run queue or the thread is dispatched directly to an idle processor (the processor is removed from the processor set's queue of ilde processors). When the kernel initiates a dispatching operation, it must first lock the thread so that the thread is not manipulated by another entity. Then the kernel takes either the **runq->lock** or the **pset->idle_lock**. (See the threads section for more details.)

This is **not** a formal hierarchy, and there exceptions and inversions that occur on a case-by-case basis. The following are some of the exceptions:

- The kernel takes **thread_lock**, then **runq->lock** when inserting a thread on the run queue. It is also possible for the kernel to take the locks in the opposite order when the thread is in a known condition (**current_thread( )**, thread is already runnable, etc.).

- The kernel takes **thread_lock**, then **processor_lock**, then **pset->idle_lock** to dispatch a thread to an idle processor. It is also possible for the kernel to take the **processor_-lock**, then call **assert_wait( ),** which in turn calls **thread_lock( )**. There is no deadlock because the **thread_lock( )** call is related to the current thread (no other thread can be the current thread).

- The relationship between **thread_lock** and **wait_lock** is interesting because they share the same level within the general hierarchy. There are two cases worth noting:

  - If the kernel needs to scan a queue of threads that are waiting on an event to make one or more of them runnable, it takes the **wait_lock** first, then takes the t**hread_-lock**(s).

  - **thread_lock** can be held across an **assert_wait( )** call (**wait_lock** and **thread_-lock**), but only if the first invocation of **thread_lock** was on a thread that is not **current_thread( )**.

8. It may happen that, in order to take locks in the proper order, the kernel must get pointers to structures that have a lower position in the hierarchy and so cannot be locked. In this case, the kernel takes the lock, then checks to make sure that conditions haven't changed. For example:

```
    pset = thread->processor_set;
    pset_lock(pset);

#if     MACH_HOST
    while (pset !=thread->processor_set) {
        pset_unlock(pset);
        pset = thread->processor_set;
        pset_lock(pset);
    }
#endif

    s = splsched( );
    thread_lock(thread);
```

9. It may happen that the kernel must acquire two locks of the same class, two task locks, for example, or two thread locks. The most common case is when the kernel must terminate a thread or a task and both the current and victim entities need to be locked. The hierarchy in this case is determined by the addresses of the entities, and, by convention, the entity at the lower address is locked first:

```
if ((int)thread < (int) cur_thread) {
    thread_lock(thread);
    thread_lock(cur_thread);
}
else {
    thread_lock(cur_thread);
    thread_lock(thread);
}
```

The kernel locks pairs of processor sets, pairs of tasks, and pairs of threads using this rule.

2.1 Tasks

The locking protocol with respect to tasks follows the general rules listed above.

A task's itk_lock proctects the structure's fields that are maintained by the IPC subsystem. These fields include **itk_self, itk_sself, itk_bootstrap, itk_exception,** and **itk_registered.** The fields protected by **task_lock** do not intersect with the fields protected by **itk_lock**. The example below shows a typical interaction between the **itk_lock** and the **ip_lock**, the IPC subsystem's most general lock:

```
itk_lock(task);
assert(task->itk_self != IP_NULL);

if((port = task->itk_sself) == task->itk_self) {
    /* no interposing */

    ip_lock(port);
    assert(ip_active(port));
    ip_reference(port);
    port->ip_srights++;
    ip_unlock(port);
} else
    port = ipc_port_copy_send(port);
itk_unlock(task);
```

2.1 Threads

The locking protocol for threads is intertwined with the protocol for the scheduler, and several exceptions to the general protocol occur; these exceptions are based on the particularities of a thread's state or the state of a queue the thread is on.

The scheduler does not distinguish between runnable threads and running threads, except for the thread executing on the current processor (**current_thread**( )). Every processor records the address of the thread it is currently executing in an entry of the **active_threads[MAXCPUS]** table, but this information can be used reliably only as **active_threads[my_processor]**; the **active_threads[other_processor]** form might be used as a hint in machine-dependent code (e.g., TLB shootdown), but is **not** used in thread management.

Consequently, the action taken by **current_thread( )** on a victim thread whose state has been set to TH_RUN must account for the following distinct situations:

- The victim thread is on a run queue.
- The victim thread is currently running.
- The victim thread is in the process of taking a processor or yielding a processor.

These three cases are important for explaining the interaction between the **thread_lock** and the **runq->lock:**

1. To insert a thread into a run queue, the kernel must lock the thread, regardless of whether the thread is coming from a wait queue or from executing (the common routine is **thread_-setrun**( )). The kernel takes **thread_lock** first, then **runq->lock**, then places the thread on the queue, setting the thread's **runq** field in the process.

2. When the kernel removes a thread from the run queue in order to execute the thread, the kernel must acquire **runq->lock** and the thread's **runq** field must be set to RUN_-QUEUE_NULL (under the **runq->lock**). As the kernel removes the thread from the run queue, the thread will not appear to have changed state to any other entity that may be referencing the thread; the algorithm will handle the fact that the thread is either runnable or running.

3. To remove another thread from a run queue, the current thread must take the victim thread's **thread_lock** before invoking **runq->lock**. After taking the **runq->lock**, the current thread checks the victim thread's **runq** field; if the field is set to RUN_QUEUE_NULL, the thread has already been extracted from the run queue. This is an instance of case 2: the kernel removed the victim thread from the run queue to execute it, and did not have to lock the victim thread to do so. In fact, the kernel may have removed the victim thread from the run queue *after* the current thread locked the victim thread but before the current thread locked the run queue.

   If the victim thread is in this state (the current thread holds the victim's lock, the victime is runnable and not on a run queue), the current thread knows that the victim is about to take a processor. The current thread knows that the victim has not just given up the processor because the current thread was able to lock the victim. (If the kernel had just removed the victim from the processor, the kernel would be holding the victim's lock as described in case 1.) Consequently, the current thread may safely release the **runq->lock** because the kernel is not in the process of inserting the victim thread back into the run queue. In summary, if the victim thread's **runq** field is set to RUN_QUEUE_NULL, it is either running or about to be run.

4. The behavior outlined in case 3 allows the kernel to avoid races when inserting or removing threads from run queues. Below is typical code that is used to handle such operations. The code is taken from **thread_dowait**( ):

```
s =splsched( );
thread_lock(thread);

for(;;) {
    case TH_RUN | TH_SUSP:
        /*
         * The thread is interruptible. If we can pull it
         * of its run queue, stop it here.
         */
        if (rem_runqu(thread) != RUN_QUEUE_NULL) {
            thread->state &= TH_RUN;
            need_wakeup = thread->wake_active;
            thread->wake_active = FALSE;
            break;
```

```
            }
#if NCPUs > 1
      /*
       * The thread must be running, so make its processor execute
       * ast_check( ). This should cause the thread to take an ast
       * and context switch to suspend for us.
       */
      cause_ast_check(thread->last_processor);
#endif NCPUS > 1
      /*
       * Fall through to wait for thread to stop.
       */
```

2.1 Additions to the General Protocol

The following list describes additions to the general protocol of the subsystem that are used in thread management:

- In addition to run queues, threads may be linked into wait queues and per port queues. To scan a wait queue, the kernel must lock the queue's head (one simple lock per hash bucket); to remove or insert a thread to or from the wait queue, the kernel must lock the thread (while still holding the wait queue lock) In contrast to run queue operations, transferring a thread to or from a wait queue causes a transition in the scheduling state machine.

- The kernel can manipulate the machine-dependent fields in the thread structure (**pcb**, **kernel_stack**) without locking the thread because these fields are manipulated only during context switches. On many platforms, kernel stack corruption is prevented by using the interrupt stack as an intermediate step in the context switch (between **save_context( )** and **load_context**( )).

- The same principal applies to those fields that are used to save information when the stack goes away.

- The kernel does not need to lock a thread when writing fields which are written only by **current_thread**( ) (e.g., **thread_privilege**, **vm_privilege**).

- As **thread_wakeup( )** locks the thread which was waitng on the specified event, the kernel must be sure not to hold such a lock before entering the call. Similarly, **thread_sleep( )** cannot be called while holding the **thread_lock** of **current_thread**( ): the lock would be released before the call to **thread_block( )**, but after the call to **assert_wait( )**; **assert_wait( )** needs to lock the thread before placing the thread on the event queue.

- Regardless of a thread's reference count, a thread always has two implicit references: the task structure holds an implicit reference by holding the head of the task's thread queue, and the processor set structure holds an implicit reference by holding the head of the processor set's thread queue. The kernel handles this as follows: after the thread's reference count has dropped to 0, it is incremented temporarily to 1, the thread's lock is dropped, then the task lock and the processor set lock are taken in the correct order (and eventually the thread lock is taken again).

- After a thread has been stopped via a call to **thread_dowait( )**, those fields within the thread that are normally modified only by the **current_thread( )** can be modified by the thread that called **thread_dowait( )** without that thread having to lock the victim. However, with respect to the victim thread's other fields, the reference count still holds and the thread must be locked before any of those fields can be modified.

- The AST handler commonly inspects thread structure fields without locking the thread; these fields include **state**, **sched_pri**, and **csw_needed**. However, in order to take action on the thread, the AST handler must lock the thread. For example, before the AST handler calls **thread_halt_self( )** after checking the thread's **thread_should_halt** field, it must first lock the thread.

- The booleans **may_assign** and **assign_active** mimic the behavior of a complex blocking lock for thread assignment operations. The call to **thread_freeze( )** will block until **may_assign** is set to TRUE; before blocking, **thread_freeze( )** sets the **assign_active** field to indicate that the thread is willing to change its assignment. Therefore, the proper sequence of calls to perform a thread assignment operation is as follows:

  thread_freeze( )

  thread_doassign(...)

  thread_unfreeze( ).

- A thread's **ith_lock** protects only those fields that are maintianed by the IPC subsystem. These include **ith_self**, **ith_sself**, **ith_exception**, **ith_reply, ith_mig_reply**, and **ith_rpc_reply.** There is no intersection between the fields protected by the **ith_lock** and the fields protected by the **thread_lock**.

2.1 Scheduling

The interaction between threads and run queues has been discussed above; in this section we focus on the scheduler's other major entities: processors and processor sets.

Processor Sets

The processor set structure is always assumed to be shared by its processors, in much the same way that the **task** structure is assumed to be shared by its threads. The **processor_set** structure does not participate in any optimizations or deviations from the general protocol outlined above.

In addition to its run queues, each processor set owns the head of three queues: a queue of assigned processors, a queue of assigned tasks, and a queue of assigned threads. The rules for queue handling described above hold for these queues and the kernel invokes **pset_lock** before iterating through them. Other rules come into play when an operation iterates through the queue AND manipulates individual elements.

The processor set structure contains the head of a fourth queue: the list of all currently idle processors (**idle_queue**). This queue is protected by the **pset->idle_lock**. This lock also protects part of the processor structure, and its behavior with respect to the processor structure is discussed in the following section.

Each processor set is entered in the kernel's queue of all processor sets. That queue is protected by the **all_psets_lock**. The kernel increments a processor set's reference count to prevent the set's being removed when the kernel drops the **all_psets_lock**.

In addition to the **pset_lock** and **pset->idle_lock**, the processor set structure contains another simple lock, the **pset->quantum_adj_lock**. This lock is invoked when the kernel needs to changes the value of the processor set's **quantum_adj_index** field. This operation is always performed in the interrupt stack by the clock interrupt handler.

Processors

The locking of the **state** field within the processor structure must be discussed because the value of that field determines the sequence in which locks must be taken. If the processor's state is PROCESSOR_OFF_LINE, PROCESSOR_SHUTDOWN, or PROCESSOR_ASSIGN, the kernel must acquire a processor's lock before performing an operation on the processor. These operations include starting a CPU and stopping a CPU. The kernel must invoke a **pset_lock** to insert a new CPU into a processor set when making the CPU available to the system.

If a processor's state is either PROCESSOR_RUNNING, PROCESSOR_IDLE, or PROCESSOR_DISPATCHING, the kernel considers the processor to be operational, and the processor's state will be managed by the scheduler. The scheduler must take the **pset->idle_lock** before changing a processor's state.

More precisely, when the kernel removes the first processor from the **pset->idle_queue**, the kernel must first take the **pset->idle_lock** before changing the processor's state from PROCESSOR_IDLE to PROCESSOR_DISPATCHING. Then the idle thread exits from its tight loop because it has found a thread to run; the idle thread changes the processor's state from PROCESSOR_DISPATCHING to PROCESSOR_RUNNING without taking any other locks.

Consequently, if we want to know the exact state of a processor, and we want to prevent the processor from changing state, we have to get the **processor_lock** and **pset->idle_lock** in the following order:

```
s = splsched( );
processor_lock(processor);
pset = processor->processor_set;
simple_lock(&pset->idle_lock);

/*
* If the processor is dispatching, let it finish - it will
* set its state to running very soon.
*/
while (*(volatile int *)&processor->state == PROCESSOR_DISPATCHING)
    continue;
```

The processor is completely under our control and its state is known only after this code has executed. The code must loop while the processor's state is set to PROCESSOR_DISPATCHING because the transition from dispatching to running occurs without any locking (it is affected by the idle thread).

This locking protocol protocol was designed to optimize the speed of the most frequent processor state transition path (take the first processor in the processor set's idle queue and set it as running). This optimization has been paid for with a sub-optimal sequence for operations that seldom occur and do not require a high degree of responsiveness. These operations include stopping a processor and reassigning a processor.

Just as we make a processor running by taking the **pset->idle_lock** and not taking the processor lock, we place a processor on the processor set's idle queue by taking the **pset->idle_lock** and not taking the processor lock.

An interesting locking protocol has been designed to handle the reading and writing of the **processor->processor_set** link. The protocol is implemented to optimize one of the most frequently travelled paths in the scheduler: a thread currently executing on the processor must be able to read the **processor->processor_set** link without locking, and then assumes that the processor does not move as the thread performs subsequent operations.

The unique role of the current processor helps. It is given the privilege of reading and writing the **processor->processor_set** field; all other processors are restricted to reading only. If another processor wants to read the field and take actions based on the value read, the processor must take the **pset_lock** for writing (although there is only one potential writer, the current processor). Consequently, the code used to move a processor to a new processor set must, at some point in the transition process, execute on the processor that is being moved:

```
this_thread = current_thread( );
thread_bind(this_thread, processor);
thread_block((void (*)()) 0);
```

3 IPC

The following analysis is limited to the case of local IPC; the code that extends IPC semantics across inter-node connections (the NORMA IPC code) is not MP safe.

The entire locking protocol of the IPC subsystem uses only simple locks. For one class of locks, the code maps calls to read_lock and write_lock into a simple lock call. Because of this mapping, it is easy to upgrade that class of locks from a simple lock mechanism to a complex lock mechanism.

The locks used in this subsystem's protocol are listed below. For each lock we provide a short description of the lock's purpose and specify the syntax used to invoke the lock.

imq_lock Protects the **ipc_mqueue** structure; this structure contains the head of the queue of kmsgs and the head of the queue of threads waiting to receive messages from the port. The lock is invoked as follows:

imq_lock(*mq***)**

Defined as follows:

#define imq_lock(mq) simple_lock(&(mq)->imq_lock_data)

io_lock Protects the **ipc_object** structure; **ip_lock** and **ips_lock** are macros that are used to invoke **io_lock** from higher level entities (namely the port and port set structures). The **io_lock** is invoked as follows:

io_lock(*io***)**

The lock is defined as followsL

#define io_lock(io) simple_lock(&(io)->io_lock_data)

The definitions for **ip_lock** and **ips_lock** are as follows:

#define ip_lock(port) io_lock(&(ipc_port_t)->ip_object)

#define ips_lock(port) io_lock(&(ipc_pset_t)->ips_object)

ipc_port_multiple_lock

Allows the calling thread to lock more than one port. This lock is invoked only once within the kernel to check for circularity problems in the following chain:

port->ip_destination->. . . == port

Within the subsystem's hierarchy, this lock must be taken before **io_lock**. The lock is invoked as follows:

ipc_port_multiple_lock( )

It is defined as follows:

#define ipc_port_multiple_lock()
simple_lock(&ipc_port_multiple_lock_data)

ipc_port_timestamp_lock_data

Used only to retrieve a monotonic timestamp value. The lock is invoked as follows**:**

ipc_port_timestamp_lock_data()

The lock is defined as follows:

**#define ipc_port_timestamp_lock() **
simple_lock(&ipc_port_timestamp_lock_data)

is_read_lock, is_write_lock

Despite the read and write "labelling," both of these locks use a simple lock to protect most of the **ipc_space** structure. They are invoked as follows**:**

is_read_lock(is)

is_write_lock(is)

The locks are defined as follows:

#define is_read_lock(is) simple_lock(&(is)->is_lock_data)

#define is_write_lock(is) simple_lock(&(is)->is_lock_data)

is_ref_lock_data

Protects the **ipc_space** structure field **is_references**. The lock is invoked only in **ipc_space_reference_macro** and **ipc_space_release_macro**. The lock is usually invoked while the calling thread holds the **is_write** lock and/or the **ip_-lock.**

ihgb_lock Protects the hash queue that is used to provide fast lookup within the subsystem's translation layer. It may be invoked while the calling thread holds an **is_read_lock** or an **is_write_lock**. The lock is invoked as follows:

ihgb_lock(ihgb)

The lock is defined as follows**:**

```
#define   ihgb_lock_init(ihgb) \
          simple_lock_init(&(ihgb)->ihgb_lock_data)
```

imarb_lock Protects the **ipc_marequest_bucket** structure; this structure contains the head of the queue of kmsgs and the head of a queue of msg-accepted requests. The lock may be invoked while the calling thread holds an **is_read_lock** or an **is_write_lock.** The lock is invoked as follow**s:**

imarb_lock(imarb)

It is defined as follows:

```
#define   imarb_lock(imarb)                    \
          simple_lock(&(imarb)->imarb_lock_data)
```

None of the locks listed above require **spl( )**.

IPC Translation Layer

A major aspect of the IPC subsystem is its translation layer, where user references to port rights are translated to their corresponding port structures. All user-invoked port management operations must be pass through this layer, including operations invoked by Mach system calls, and operations invoked via the transfer of port rights in IPC messages.

When a translation operation is underway, the kernel locks the translation table with either the **is_read_lock** or the **is_write_lock**. The kernel holds the lock until the operation is complete. Locking the translation table via **is_write_lock** indicates that fields within the ipc space structure are going to be changed; therefore, a complex lock, if used, must be held in an exclusive way.

If the operation is translating a port right to a pointer to the corresponding port data structure (e.g., the user has issued a call to **mach_msg_send( )**), the kernel holds the ipc space lock until it is able to lock the underlying port structure (in the simplest case, holding **is_read_lock** is sufficient as no data is changed).

If the operation involves the translation of a port structure to a port right, (e.g., the user has invoked **mach_msg_receive( )** and the incoming message contains port rights that the kernel will install in the user's IPC space), again the kernel must lock lock the IPC name space before it locks the port structure. The kernel must invoke **is_write_lock** before installing port rights in an IPC name space.

Note that an IPC name space may be addressed in the following ways: through the task structure's **itk_space** field (either as **current_thread( )->task->itk_space** or as **((task_t)port->ip_kobject)->ikt_space**) or through **port->ip_receiver**.

io_lock, **ip_lock**, and **ips_lock** are three forms that lead to the same simple lock, which is located in the **ipc_object** structure. This simple lock protects the **ipc_object** structure and many of the higher level abstractions, including the port kernel structure (**ipc_port**) and the port set structure (**ipc_pset**). The **ipc_object** structure is embedded at the top of both the port and port set structures and so it is common practice to access the **ipc_object** (and the corresponding **io_lock**) by recasting a reference. In this way, a referencing thread need not care about whether or not the higher level structure is a port or a port set.

Ports and Port Sets

Both the port and port set structures include two queues: a queue that contains messages waiting for delivery, and a queue of threads waiting to receive messages. These queues are protected by the port or port set structure's **imq_lock**.

Although the port and port set structures include the same class of locks (**io_lock** and **imq_lock**), there is an order that relates the port locks with the port set locks. The port structure has a higher position in the hierarchy and so the following two sequences are allowed:

With respect to the **ip_lock** and **ips_lock**:

```
ip_lock(port);
pset = port->ip_pset;
if (pset !=IPS_NULL) {
    ips_lock(pset);
    . . . . . . .
```

With respect to the **imq_lock**:

```
imq_lock(&port->ip_messages);
imq_lock(&pset->ips_messages);
```

General Hierarchy

Having introduced a set of dependencies above, we can now summarize the hierarchy of the IPC subsystem as follows:

higher position	is_read_lock, is_write_lock, ipc_port_multiple_lock
-	ip_lock (eq. to io_lock(&ipc_port_t)->ip_object)
-	ips_lock (eq. to io_lock(&ipc_pset_t)->ips_object)
-	simple_lock(&(is)->is_ref_lock_data)
-	imq_lock(&port->ip_messages)
lower position	imq_lock(&pset->ips_messages)

The kernel uses conditional locking whenever the kernel needs to lock multiple objects that are at the same level in the hierarchy. In fact, within this subsystem, the forms **ip_lock_try** and **imq_lock_try** are often invoked in order for the kernel to decide whether to follow the most general algorithm, or to follow the optimized path, based on an atomic action upon multiple ports.

The order of taking locks follows the hierarchy outlined above, but the order in which locks are released does not. Generally, the kernel releases a given lock as soon as it has located and locked the next abstraction in the code path. This is to be expected, given the nature of most IPC code paths, which must cross the IPC space translation layer, then the port structure layer, then the queue management layer. The design of locked sections with minimum overlap provides the highest degree of parallelism.

The most general locking approach that can be represented for the IPC subsystem is as follows:

```
is_read_lock(space);
    ......
/* find the port matching the user port right */
    ......
ip_lock(port);
is_read_unlock(space);
    ......
imq_lock(&port->ip_messages);
ip_unlock(port);
    ......
/* queue the message */
    ......
imq_unlock(&port->ip_messages);
```

An operation that changes an IPC name space (**is_write_lock** is required) must hold the IPC space lock while it performs all or part of the port kernel structure processing, because the result of this last phase will be used to write into the IPC name space.

Once a message queue is locked at the bottom of a transaction, the message queue’s port can be unlocked because we know the port is active and the message in the queue holds a reference to the port.

Across this path, the kernel uses the reference counts of the IPC name space and of the port structure to prevent these structures from being destroyed. The system provides macros that allow a thread to check when a lock is released to determine if the object’s reference count has dropped to 0 and should be released as well. For example:

```
#define io_check_unlock(io)
MACRO_BEGIN
    ipc_object_refs_t _refs = (io)->io_references;

    io_unlock(io);
    if(_refs == 0)
```

```
        io_free(io_otype(io), io);
MACRO_END
```

Most of the procedures implemented by the IPC subsystem affect the locking state of the entities they manipulate. Some procedures return locked resources to their callers; others release locks that the caller had taken before the call. Some procedures return locked resources if successful and unlocked resources if they are not successful.

A significant part of the locking design is affected by the demand for atomicity. For example, a given procedure may want to manipulate multiple port structures and install or remove multiple port rights without interference from other procedures. More precisely, we distinguish the cases of atomic copyin and atomic copyout. The copyin operation copies a message into kernel format, translating the message's port rights (if any) to their port structures. The copyout operation copies a message from kernel format into the user's address space when the user invokes a receive operation, installing or incrementing reference counts on port rights that may be contained in the message.

atomic copyin
: is based on locking the IPC space in read mode, locking the **dest_port** and (conditionally) locking the **reply_port**. Once both ports are locked, the copyin operation can release the IPC space lock.

atomic copyout
: is based on locking the IPC space in write mode, locking the **dest_port** and locking (conditionally) the **reply_port.** The **reply_port** is locked only to check that it is active at the time that the **dest_port** is known to be active (the two ports must be active at the same time). **is_write_lock** is taken until the point at which the user-level port right has been generated for the **reply_port** and is installed into the IPC space.

If the operation is unable to take the conditional lock on the second port, the algorithm assumes that it is not possible to lock both ports at the same time. Therefore, the algorithm must be able to handle the case that one or both of the ports has been destroyed between the copyin/copyout of the first port and the copyin/copyout of the second port. Without going into all the details of the recovery algorithm the basic approach is as follows (copyin operation):

- The algorithm acquires the write lock for the IPC name space.
- It copies in the **dest_port**.
- The algorithm gets a pointer to the **reply_port** (the port is not copied in at this time) and increments the port's reference count if the port is alive and copies the port.
- The algorithm copies in the **reply_port**.
- If the reply port has died in the meantime, the algorithm checks the saved copy and examines its timestamp. If the **dest_port** has also died in the interim, and if it died before the **reply_port** died (the algorithm can tell by comparing timestamps), both copyins must be undone. This is because the copy operation indicated that the **dest_port** is alive and the **reply_port** is dead, and this is exactly the opposite of what happened.

Interactions with the Task and Threads Subsystem

The IPC subsystem interacts with the task and threads subsystem through a set of fields that are contained in the task and thread data structures. The following lists the fields and describes how they are used:

- The task structure's **itk_space** field points to the task's IPC space. The kernel may access the IPC space from the task's side without locking. (The pointer is set once, when the task is created).

- The **itk_self**, **itk_sself**, **itk_bootstrap**, **itk_exception**, and **itk_registered[ ]** fields describe the set of IPC resources that every task owns by default. These fields are protected by the **itk_lock**; this lock is taken before the **ip_lock** is taken for operations that manipulate individual IPC resources.

- The thread structure's **ith_next, ith_prev, ith_state, ith_seqno,** and **ith_messages** fields are used by the IPC subsystem to manipulate threads in various ways. The subsystem uses the **ith_next** and **ith_prev** queues when placing or removing a thread from a port thread queue such as a port's queue of blocked threads (**(ipc_port_t)->ip_blocked**) or a port's queue of ready-to-receive threads (**(ipc_port_t)->ip_messages.imq_threads**). The subsystem uses the **ith_state, ith_seqno,** and **ith_messages** fields to manipulate a thread's IPC state.

  The kernel may manipulate these fields without taking any locks as only two cases apply: the operation is taking place in the context of the current thread, or the IPC subsystem is removing the thread from one of the IPC queues. Queueing and dequeueing take place under the protection of the lock associated with the specific queue.

- The thread structure's **ith_self**, **ith_sself**, **ith_exception**, **ith_reply**, **ith_mig_reply**, and **ith_rcp_reply** fields specify the set of IPC resources that every thread inherits by default. These fields are protected by the thread structure's **ith_lock**; this lock is taken before the **ip_lock** is taken for operations that manipulate individual IPC resources.

- The thread structure's **ith_msize, ith_kmsg**, and **ith_wait_result** fields specify various IPC attributes associated with the thread. The kernel may manipulate these fields without taking any locks as only two cases apply: the operation is taking place in the context of the current thread, or the IPC subsystem is removing the thread from one of the IPC queues.

3.1 Port and Port Set

In addition to the general rules described above, the following sequences, related to ports and port sets, appear frequently in the code:

- The following port fields -- **ip_mscount**, **ip_nsrequest**, **ip_pdrequest** -- are usually read within a locked section (**ip_lock**). If the condition tested within the locked section becomes true (e.g., a port's **ip_srights** drops to 0), the lock is released and the saved value can be used for further processing, a no-senders notification, for example. The sole requirement is that the value read (how many send rights were created) must be consistent with the condition being true (e.g., a port's **ip_srights** has dropped to 0).

- If an **ip_lock** must be dropped prematurely because of a call that may block, the kernel must take a reference on the port; the reference is released when the kernel relocks the port.

- There are several optimizations that reduce the critical sections that require locks. There are also cases where the critical section can be avoided all together. For example, the **(ipc_port_t)->msgcount** is lazily updated. Therefore, the subsystem skips incrementing/decrementing and all queueing operations whenever it is possible to handoff a message from sender to receiver.

- In order to move a port from one port set to another, a number of locks must be acquired to provide atomicity. Nonetheless, hierarchy helps. The IPC space is read locked and the target port is locked. If both the old port set and the new port set are active, the operation must acquire the locks to both port sets. The order in which the locks are acquired are based on the port set addresses:

```
if (oset < nset) {
    ips_lock(oset);
    ips_lock(nset);
} else {
    ips_lock(nset);
    ips_lock(oset);
}
```

  The reference count mechanism also comes into play to prevent the target port and both port sets from being destroyed. The IPC name space holds references to the target port and the new port set, and the target port holds a reference to the old port set.

4 Virtual Memory

In addition to analyzing the machine-independent hierarchy of the VM subsystem, this study also analyzes zone module and the machine-dependent PMAP module. Although the PMAP module is largely machine-dependent, we believe that different implementations of the module have very similar locking protocols. The TLB shootdown algorithm is the sole exception, and a notable one at that.

This subsystem employs a range of lock types in its locking protocol. Each address space has an associated complex locking block. This lock sits at the top of the hierarchy. The subsystem maintains a system-wide non-blocking complex lock that is used near the bottom of the VM processing. The subsystem also employs several simple locks. Some of these are system-wide locks, and the others are associated with individual structures.

The locks used in this subsystem's protocol are listed below. For each lock we provide a short description of the lock's purpose and specify the syntax used to invoke the lock.

vm_map_lock Protects the **vm_map** and **vm_map_entry** structures. This lock is implemented as a complex lock and so has two forms: **vm_map_lock_read** locks the structures for reading, **vm_map_lock** locks the structures for writing. The locks are invoked as follows:

vm_map_lock_read(*map***)**

vm_map_lock(*map***)**

The locks are defined as follows:

```
#define   vm_map_lock_read(map)   lock_read(&(map)->lock)
#define   vm_map_lock(map)
MACRO_BEGIN
    lock_write(&(map)->lock);
    (map)->timestamp++;
MACRO_END
```

vm_map->ref_lock

This lock protects the **vm_map** structure's **ref_count** field. The lock is invoked as follows:

simple_lock(&(vm_map_t)->ref_lock)

Separate locking interface macro not defined.

vm_map->hint_lock

This lock protects the **vm_map** structure's **hint** field. The lock is invoked as follows:

simple_lock(&(vm_map_t)->hint_lock)

Separate locking interface macro not defined.

vm_object_lock

Protects most of the **vm_object** structure, and is also used during certain operations on pages. The lock is invoked as follows:

vm_object_lock(*object***)**

The lock is defined as follows:

```
#define vm_object_lock(object) \
MACRO_BEGIN \
    simple_lock(&(object)->Lock); \
    (object)->LockHolder = (int) current_thread(); \
MACRO_END
```

vm_object_cache_lock

Protects the queue of cached VM objects and also protects certain fields within the VM object structure (fields related to IPC ports). The lock is invoked as follows:

vm_object_cache_lock( )

The lock is defined as follows:

**#define vm_object_cache_lock( ) **
simple_lock(&vm_object_cached_lock_data)

vm_page_bucket->lock

Protects a hash bucket within the VM subsystem's object/offset page table. There is one of these locks per hash bucket within the page table. The lock is invoked as follows:

simple_lock(&(vm_page_bucket_t)->lock)

Separate locking interface macro not defined.

vm_page_lock_queues

Protects the active and inactive page queues, plus some fields within the **vm_page** structure. The lock is invoked as follows:

vm_page_lock_queues( )

The lock is defined as follows:

**#define vm_page_lock_queues( ) **
simple_lock(&vm_page_queue_lock)

vm_page_queue_free_lock

Protects the free page and fictitious page queues. The lock is invoked as follows:

simple_lock(&vm_page_queue_free_lock)

Separate locking interface macro not defined.

memory_manager_default_lock

Serializes operations on the default memory manager (set and reference the default memory manager). The lock is invoked as follows:

simple_lock(&memory_manager_default_lock)

Separate locking interface macro not defined.

zone->complex_lock

Protects the **zone** structure of a pageable zone. The lock is invoked as follows:

lock_write(&(zone_t)->complex_lock)

Separate locking interface macro not defined.

zone->lock Protects the **zone** structure of a non-pageable zone. The lock is invoked as follows:

simple_lock(&(zone_t)->lock)

Separate locking interface macro not defined.

zget_space_lock

Protects the common heap (**zalloc_next_space, zalloc_end_of_space**) that memory is taken from to expand non-pageable zones. The lock is invoked as follows**:**

simple_lock(&zget_space_lock)

Separate locking interface macro not defined.

zone_page_table_lock

simple_lock(&zone_page_table_lock)

Separate locking interface macro not defined.

all_zones_lock

simple_lock(&all_zones_lock)

Separate locking interface macro not defined.

PMAP_COMPLEX_LOCK

A complex non-blocking lock that prevents deadlocks within the PMAP module. The lock is invoked as follows:

lock_read(&pmap_system_lock)

lock_write(&pmap_system_lock)

Separate locking interface macro not defined.

PMAP_SIMPLE_LOCK

Protects the pmap structure. The lock is invoked as follows:

simple_lock(&(pmap_t)->lock)

Separate locking interface macro not defined.

lock_pvh_pai Protects the **pv_list**, a queue of pointers to pmaps, maintained on a per physical page basis. The lock is invoked as follows**:**

lock_pvh_pai(*pai***)**

Defined as follows:

#define lock_pvh_pai(pai) (bit_lock(pai, pv_lock_table))

pv_free_list_lock

Protects the free list that the elements of the **pv_list** are allocated from and returned to. The lock is invoked as follows:

simple_lock(&pv_free_list_lock)

Separate locking interface macro not defined.

pmap_update_list->lock

Protects the control block used within the TLB shootdown algorithm. The lock is invoked as follows:

simple_lock(&(pmap_update_list_t)->lock)

Separate locking interface macro not defined.

Masking Interrupts

The following set of locks can be acquired only at **splvm( )** priority; on most platforms, this corresponds to a priority of **splhigh( )**:

PMAP_COMPLEX_LOCK

This lock is invoked within the TLB shootdown protocol and is invoked within interrupt handlers.

PMAP_SIMPLE_LOCK

This lock is invoked within the TLB shootdown protocol and is invoked within interrupt handlers.

lock_pvh_pai This lock is invoked within the TLB shootdown protocol and is invoked within interrupt handlers.

pmap_update_list->lock

This lock is invoked within the TLB shootdown protocol.

General Hierarchy

Much of the locking protocol design is based on the features provided by the **vm_map** complex lock, and many of the algorithms within the VM subsystem take advantage of this lock's features which include the following: multiple readers/single writer, read-to-write and write-to-read upgrades, and recursive locking. The **vm_map** lock is the only lock in the current system that provides these capabilities. Study of this lock may motivate the use of its features in other parts of the system in the future.

The **vm_map** lock is the coarsest lock in this subsystem, and it can be held across context switches. It protects a task's **vm_map** structure and the corresponding **vm_map_entry** structures. Also note the existence of sub-maps within the kernel's **vm_map**. Sub-maps are also protected by **vm_locks**.

Each **vm_object** structure is protected by a simple lock; this lock covers all fields except the object's IPC-related fields. The IPC-related fields are protected by the system-wide **vm_object_cache_lock** and cannot be accessed unless this lock is acquired.

The **vm_page** structure does not have its own lock. When such a structure is allocated to a VM object, it is protected by that VM object's lock. **vm_page** structures are also protected by the page queue locks. Depending on the operation, a **vm_page** structure's fields may be safely manipulated if the corresponding **vm_object** lock is held, if and associated page queue lock is held, or if both locks are held.

At the bottom of VM subsystem, the PMAP module handles all machine-dependent operations. The PMAP module establishes its own, internal hierarchy; this hierarchy does not in any way depend upon the hierarchy inherent in the machine-independent VM (henceforth referred to as the general VM hierarchy). However, the general VM hierarchy does depend to some extent on the PMAP hierarchy, because the machine-independent VM must be aware of all PMAP operations that may potentially block.

The following describes the VM subsystem's locking hierarchy:

higher position	vm_map_lock, ip_lock
-	vm_object_cache_lock
-	vm_object_lock
-	vm_page_lock_queues
lower position	PMAP hierarchy, other simple locks without dependencies

Note that multiple entries at a level (for example vm_map_lock and ip_lock) do not depend on one another.

Besides the above-mentioned recursive use of locks, the subsystem frequently uses conditional locking constructs. These are used in operations where locks at the same level must be acquired, or when locks must be acquired in inverted order.

Protocol Optimizations

Typical optimizations to the subsystem's locking protocol are as follows:

- Newly created entities are manipulated without being locked. This includes new addresses that have been created but not yet given to the user. These entities can be safely manipulated because they are only known by the thread that has just created them.

- The reference count mechanism, as in the other subsystems, is used to indicate when a given entity can be deallocated and garbage-collected. Both of these operations can be invoked without the taking of locks

- Boolean flags mimic the behavior of complex blocking locks: if the entity is found with the flag turned on, a number of operations must be delayed and the calling thread is blocked to wait for the flag to be reset. An example of this is the **vm_object** flag **paging_in_progress**. In contrast to blocking locks, which block all operations, the boolean flag mechanism blocks specific operations and allows other operations to continue.

- The subsystem avoids various races by introducing special entities at the beginning of a queue to prevent other processors from accessing the same queue and attempting to duplicate operations that are already underway. For example, the subsystem uses *fictitious* pages to prevent multiple processes from attempting to pagein the same page.

- Although the kernel's address space is implemented via the same data structures used to implement user address spaces (an exception is the submap abstraction, which is used only in the kernel map), the locking protocol can be optimized for operations on the kernel map or the kernel PMAP because the kernel can assume that certain operations (for example **vm_map_destroy( )**) will not be performed on the kernel map or the kernel PMAP.

- Timestamps are used to avoid redoing lookup operations when a lock has been dropped, then reacquired. In most cases, the degree of contention is not high and the timestamp mechanism allows a thread to proceed with its operations without having to perform a needless lookup operation.

As an implementation detail, many of the VM routines have return values that indicate locking state. For example, **vm_map_lookup( )** performs the following operations:

1. The routine acquires a **vm_lock** (the mode of which depends on the parameters passed to the routine).

2. The routine acquires a **vm_object_lock**.

3. The routine sets the **vm_map**'s timestamp and releases the **vm_map_lock**.

4. The routine returns the locked **vm_object** and the **vm_map** timestamp.

4.1 vm_map

The **vm_map_lock** is often held while the current thread waits on an event. Here are the possible scenarios:

1. While holding a **vm_map_lock**, the thread attempts to lock another **vm_map**, but that lock is busy. This is a situation that occurs when the thread is attempting to modify a kernel submap (which is equivalent to a common map). A typical example is when a call to **zalloc( )** finds that the zone is empty and attempts to expand it, but another allocation from the zone is in progress and so the submap for the zone is already locked. A similar situation can arise with the **kalloc_map**. Excluding these submap examples, a thread never needs to hold two (or more) **vm_map** locks at a time. The classic case of **fork( )** is solved in that only the parent's **vm_map** must be locked. The child's map does not need to be locked because it is newly created and cannot be referenced by other threads.

2. As a more general expression of case 1, a thread may hold a **vm_map_lock** and then issue a call that blocks. The most typical example of this is when a sequence of calls end in a request for zone expansion (e.g. **pmap_enter( )** at the bottom of a page fault path). The thread must block while waiting for pages to become available, and the thread must poke the pageout thread and wait for that thread to free pages.

Note that the locking protocol does not require a thread to hold the **vm_map_lock** in the pageout path while the thread waits for the pager to supply the data. This common case is handled as follows: the kernel allocates a fictitious page, marks the page busy, and inserts it into the top VM object in the shadow chain. Other threads that fault on the same page find this busy page and block to wait for the current pagein operation to complete. Although the page fault path still requires a **vm_map_lock** to read the map (write mode is required only in the case that a new VM object is to be mapped into the address space), the page fault path releases the lock while waiting for the page, then reaquires the lock when the page is available. At this point, the page fault path checks the **vm_map**'s timestamp and avoids a lookup operation if the timestamp hasn't changed.

Recursive Locking

The VM subsystem allows privileged tasks to make portions of their address spaces nonpageable, and this functionality requires the kernel to recursively lock the target **vm_map**, as follows:

1. The task initiates the operation that wires down the affected pages. Since this operation is going to modify the **vm_map**, it locks the map in write mode.

2. The operation prepares to fault in the pages, creating shadow objects as required, and so on.

3. The operation downgrades its lock on the **vm_map** to read mode and marks the lock as recursive.

4. The operation invokes the page fault handler to fault in the pages. As each page is paged in, the page fault handler must lock the map in read mode, then unlock the map. Throughout the pagein of the pages, the operation holds a read lock on the map.

5. When the pages are all in, the page fault handler releases its last read lock, then the operation releases its read lock.

It has been noted that the above mechanism could generate a deadlock if applied to the kernel map. Suppose the kernel needs to wire a set of pages and locks its map in write mode to begin the operation. Suppose that the page fault handler blocks to wait for pages to become free. As mentioned above, the page fault handler releases all locks before it blocks, including the **vm_map** lock. The wiring operation still holds a read lock on the map, and the pageout operation, which would free the resources that the page fault handler is waiting for, cannot lock the kernel map in write mode.

Consequently, the kernel does not used the recursive mechanism on its own map. Instead, a wiring operation on the kernel map drops its read lock on the kernel map before initiating the page fault cycle. The dropping of this lock assumes that kernel threads are well behaved and will not perform any destructive operations while the page fault cycle runs its course.

4.2 vm_objects and vm_pages

As mentioned above, **vm_page** structures do not have locks associated with them; operations that manipulate these structures are serialized through the system-wide page queue locks and/or the associated **vm_object_lock**. In addition to discussing the interactions between the page queue locks and **vm_object_locks**, this section discusses the interactions between the **vm_object_lock** and the system-wide **vm_object_cache_lock**.

vm_object_cache_lock

The **vm_object_cache_lock** has a higher position in the hierarchy than the **vm_object** lock. When it is invoked, the **vm_object_cache_lock** is acquired before the calling thread invokes the **vm_object_lock**. This sequence is invoked when a thread adds or removes a VM object from the kernel's object cache list.

Also, the **vm_object_cache_lock** is used to manage the IPC port association to the **vm_object** and the translation from an IPC port to a **vm_object** port, as is shown in the following code fragment:

```
vm_object_cache_lock( );
if(ip_kootype(pager) != IKOT_PAGER) {
    vm_object_cache_unlock( );
    return;
}
object = (vm_object_t)pager->ip_kobject;
vm_object_lock(object);
```

If the algorithm cannot follow this hierarchy, the conditional form (**vm_object_cahce_lock_try**) is used (e.g., **vm_object_collapse( )**).

vm_object_lock

When studying the interactions that occur when a thread must lock multiple VM objects, three distinct cases are worth noting:

1. The VM objects to be locked are in a shadow chain.
2. The VM objects to be locked are in a copy chain.
3. The VM objects to be locked are not related.

In case 1, the protocol is based on the fact that a shadow chain can be traversed in only one direction, from the newest VM object to the original VM object. This dictates a natural locking order, and usually the lock to one VM object is held until the next VM object in the chain is located and locked, at which point the first VM object's lock is released. The following code fragment illustrates this protocol:

```
next_object = object->shadow
if(next_object != VM_OBJECT_NULL) {
    offset +=object->shadow_offset;
    vm_object_lock(next_object);
    vm_object_unlock(object);
    object = next_object;
}
```

For case 2, the copy object link is a back pointer to a shadow object, and the potential deadlock is prevented by the following code:

```
if(!vm_object_lock_try(copy_object) {
    vm_object_unlock(object);

    simple_lock_pause( );  /* wait a bit */

    vm_object_lock(object);
    continue;
}
```

For case 3 (the VM objects are not related), the protocol employs conditional locking.

It must also be noted that the protocol for invoking the **vm_object_cache_lock** and **vm_object_-locks** has been carefully designed so that they are never held across blocking PMAP calls, such as **pmap_enter( )**.

Issues Related to vm_page Structures

The locking of **vm_page** structures is as follows:

- The fields used to link a given **vm_page** structure to the active, inactive, and free page lists are protected by the **vm_page_lock_queues.**
- The fields used to link a given **vm_page** structure to the VM object's queue of pages are protected by the associated **vm_object_lock**.
- The field used to link a given **vm_page** structure to the VM subsystem's object/offset hash table is protected by the **vm_object_lock**.

- The backpointer from the **vm_page** structure to the **vm_object** structure, and the offset within a **vm_object** must be set while both the **vm_object_lock** and the **vm_page_lock_queues** are held.
- To modify a **vm_page**'s **wire_count** flag, the calling thread must take the **vm_object_lock** and the **vm_page_lock_queues**.
- The fields used to specify a page's state with respect to the page replacement algorithm (the **inactive**, **active**, **free**, **laundry**, and **reference** fields) are protected by **vm_page_lock_-queues**.
- The fields used to specify a page's state with respect to its VM object (the **busy**, **wanted**, **tabled**, **fictitious**, **private**, **absent**, **error**, **dirty**, **precious**, and **overwriting** fields) are protected by the **vm_object_lock**.
- The fields used to specify a pages state with respect to being locked by the memory manager (the **page_lock** and **unlock_request** fields) are protected by the **vm_object_lock**.

The Page Queues

The kernel uses **vm_page_lock_queues** to keep the inactive, active, and free page queues consistent, and, as a consequence, to manage those **vm_page** structure fields that are associated with the management of the page queues. The **vm_page_lock_free_queue** covers the allocation of free and fictitious pages.

The subsystem maintains a set of counters that it uses in managing the page queues (**vm_page_-free_count**, **vm_page_inactive_count**, etc.). The subsystem does not attempt to keep these counters consistent with one another so the VM algorithms must assume that these counters may hold stale information. However, the sum of these counters must be invariant.

The protocol used by the system when waking up threads that are waiting for pages to become free is worth discussing at this point. The problem is as follows: when the VM subsystem releases a page (**vm_page_free**), it wakes up only the first page that is queued waiting. Suppose this thread immediately asks for a new page and goes back to sleep, remaining at the head of the queue of threads waiting for free pages. If this cycle continues, the thread will starve all the other threads. The solution is to use the **thread_wakeup_one( )** routine; if the thread immediately asks for another free page, the routine places the thread at the tail of the wait queue and other threads can access to resources.

4.1 PMAP

There are three types of operations that may be invoked when the kernel invokes the PMAP module:

1. Operations that affect only the specified pmap.
2. Operations that affect the specified pmap and the pv_list.
3. Operations that are given a physical page as a parameter (not a pmap as in cases 1 and 2) and must manipulate all pmaps that have mapped the physical page.

Operations of type 1 must invoke only the **PMAP_SIMPLE_LOCK.** Operations of types 2 and 3 must invoke **PMAP_SIMPLE_LOCK** and **lock_pvh_pai**. In the case of type 2, the **PMAP_SIMPLE_LOCK** is taken first, then the **lock_pvh_pai** is taken. In the case of type 3, the **lock_pvh_pai** is taken first, then **PMAP_SIMPLE_LOCK.**

Because type 2 and type 3 operations take these locks in the opposite order, a potential deadlock can occur. To prevent this deadlock, the **PMAP** module implements a system-wide **PMAP_COMPLEX_LOCK**, which provides an additional layer of locking. The lock's multiple readers feature is assigned to the type 2 operations because they are more frequently invoked, and the lock's single writer feature has been assigned to the type 3 operations.

The **PMAP_COMPLEX_LOCK** is not a blocking lock; therefore it can be compared to a simple lock; the only difference is that it supports multiple readers. There are several reasons why the PMAP_MODULE prevents threads from context-switching while the **PMAP_COMPLEX_LOCK** is busy. First of all, PMAP operations are invoked at the bottom of the VM processing path and it is frequently the case that the path has taken several simple locks within the machine-independent VM that the thread cannot release. Also, certain interrupt handlers issue PMAP calls (interrupt handlers which are given a physical address and need to change the page's attributes.

It is not possible for a thread to acquire multiple **PMAP_SIMPLE_LOCK**s or **lock_pvh_pai**s.

It has been said that the PMAP locks must all be taken at **splvm( )** priority because of the interaction between interrupt handlers and the TLB shootdown protocol. With respect to TLB shootdown, the role of the bitmap **cpus_active**, which is modified through atomic bitwise set/unset operations, needs t o be discussed. After raising the hardware priority, the processor must remove itself from **cpus_active** before acquiring any lock (PMAP_COMPLEX_LOCK or PMAP_SIMPLE_LOCK), because the TLB shootdown initiator, which already owns the PMAP_COMPLEX_LOCK and/or the PMAP_SIMPLE_LOCK, has posted a request to other processors and is waiting for its request to be honored. Consequently, the shootdown initiator would never see its request acknowledged because the request, in the form of an inter-processor interrupt, would never arrive, because it is blocked by the other processor spinning at **splvm( )** on the busy lock. This deadlock is prevented by the following mechanism:

1. The processor removes itself from the **cpus_active** bitmap after invoking **splvm( )** and before requesting the PMAP_COMPLEX_LOCK (or PMAP_SIMPLE_LOCK).

2. The processor also removes itself from the **cpus_active** bitmap once it is within the TLB shootdown interrupt handler.

3. The shootdown initiator waits for the TLB request to be acknowledged, spinning until all of the target processors have removed themselves from **cpus_active**.

4. In the TLB shootdown interrupt handler, right after the processor removes itself from **cpus_active**, the processor waits until both the **PMAP_SIMPLE_LOCK** of the **kernel_pmap** and the **PMAP_SIMPLE_LOCK** of the target pmap (they may be the same) become free. Then the interrupt handler performs the TLB flush operation.

5. The shootdown initiator is released from its wait as soon as the processor(s) unset **cpus_active**. The initiator releases its own locks and allows the TLB shootdown handler to perform the TLB flush operation.

Note that the TLB shootdown protocol, with the mechanism just described, also prevents other processors from making implicit changes to the page table entry attributes (e.g., the modify bit changed by the hardware upon reference) while in the process of invalidating TLBs. To prevent other processors from using a pmap for which there is a pending TLB flush, **PMAP_ACTIVATE_KERNEL** and **PMAP_ACTIVATE_USER** must acquire the **PMAP_SIMPLE_LOCK**.

Some PMAP procedures can block to wait for memory allocation, and the caller must be prepared to handle such a block. The caller must release all simple locks and decide whether to keep or to release the blocking lock (**vm_map_lock**). The PMAP calls that may block are as follows:

- **pmap_create( )**
- **pmap_expand( )** (PMAP internal use)
- **pmap_enter( )**
- **pmap_map( )** (A wrapper for **pmap_enter( )**)

Among these procedures, **pmap_enter( )** is the most critical because its blocking behavior affects the entire VM locking protocol. This call may also be called recursively, due, for example, to the allocation of pages while in the process of treating a page fault.

4.2 Zones

The locking protocol for this subsystem is very simple, as there are no instances of nested locking. It is worth noting that the **zone** structure includes both a simple lock and a complex lock. The locking protocol determines which lock to take based on whether or not the zone is pageable. If the zone is pageable, the associated structure is protected with a complex lock. If the zone is not pageable, it is protected with a simple lock.

5 Devices

The study of this subsystem is limited to the machine-independent sections. These sections include the device protocol entry points which are invoked by Mach IPC MiG-generated stubs, the device processing section which is still device-independent (character device, block device, and network device), and the device pager for memory mapped character devices. This study does not discuss the subsystem’s bottom layer, which handles devices and depends significantly on the device/IO interface.

The locks used in this subsystem’s protocol are listed below. For each lock we provide a short description of the lock’s purpose and specify the syntax used to invoke the lock. The entire locking protocol employs simple locks.

ior_lock Used to synchronize **iowait( )** and **iodone( )** calls that are executed on different processors. The lock is invoked as follows:

ior_lock(*io_req***)**

Defined as follows:

```
#define    ior_lock(io_req)  simple_lock(&(io_req_t)->io_req_lock)
```

io_done_list_lock

Protects the subsystem's **io_done_list**, a queue of requests for the invocation of completion functions. The lock is invoked as follows:

simple_lock(&io_done_list_lock)

Separate locking interface macro not defined.

dev_number_number_lock

Protects the **dev_number_hash_table**. The lock is invoked as follows:

simple_lock(&dev_number_lock)

Separate locking interface macro not defined.

device_lock Protects part of the **device** structure. The lock is invoked as follows:

device_lock(*device***)**

Defined as follows:

#define device_lock(dev) simple_lock(&(device_t)->lock)

device->ref_lock

Protects the **device** structure's **ref_count** field. The lock is invoked as follows**:**

simple_lock(&(*dev***)->ref_lock)**

Separate locking interface macro not defined.

dev_pager_hash_lock

Protects the **dev_pager_hash_table[ ]**, which is used in translating **ipc_port_t** types to their corresponding **dev_pager_t** types. The lock is invoked as follows:

simple_lock(&dev_pager_hash_lock)

Separate locking interface macro not defined.

dev_pager->lock

Protects part of the **dev_pager** structure. The lock is invoked as follows:

simple_lock(&(*dev_pager***)->lock)**

Separate locking interface macro not defined.

net_queue_lock

Protects the **net_queue_low** and **net_queue_high** queues, and the boolean **net_thread_awake**. The lock is invoked as follows:

simple_lock(&net_queue_lock)

Separate locking interface macro not defined.

net_queue_free_lock

Protects the **net_queue_free** queue. The lock is invoked as follows:

simple_lock(&net_queue_free_lock)

Separate locking interface macro not defined.

net_kmsg_total_lock

Protects the **net_ksg_total**, **net_queue_free**, and **net_kmsg_max** counters. The lock is invoked as follows:

simple_lock(&net_kmsg_total_lock)

Separate locking interface macro not defined.

if_rcv_port_list_lock

Protects the list of filters (**ipc_port_t** types) specified by **(struct ifnet *)->if_-rcv_port_list**. The lock is invoked as follows:

simple_lock(&(*ifnet* ***)->if_rcv_port_list_lock)**

Separate locking interface macro not defined.

ifq_lock Supports the queuing of **ifqueue** structures in the conventional BSD style. (**IF_ENQUEUE, IF_PREPEND, IF_DEQUEUE)**. The only difference is that **mbufs** are replaced with **io_req_t** objects. The lock is invoked as follows:

simple_lock(&(*ifnet* ***)->ifq_lock)**

Separate locking interface macro not defined.

t_lock Comparable to standard UNIX tty discipline locking. The lock is invoked as follows:

simple_lock(&(*tty* ***)->t_lock)**

Separate locking interface macro not defined.

Masking Interrupts

Given this subsystem's involvement with the device/IO interface, many of the locks listed above must be taken at some level of hardware priority. The following lists the locks and indicates what level of priority each lock must be invoked at:

ior_lock Used by device interrupt handlers, so priority must be set to **splio( )**. To mask all device interrupts, **splio( )** must be equivalent to **splhigh( )** on most platforms.

net_queue_lock
Used by network interrupt handlers, so priority must be set to **splimp( )**.

net_queue_free_lock
Used by network interrupt handlers, so priority must be set to **splimp( )**.

net_kmsg_total_lock
Used by network interrupt handlers, so priority must be set to **splimp( )**.

ifq_lock Used by network interrupt handlers, so priority must be set to **splimp( )**.

t_lock Used by character device interrupt handlers, so priority must be set to **spltty( )** .

The machine-independent section of this subsystem is designed to be general enough to incorporate all possible device/IO interfaces (these interfaces are implemented in the machine-dependent section of the subsystem). The machine-independent locking protocol makes no assumptions about the locking protocol of the machine-dependent section, and paths through the subsystem do not hold locks across the machine-dependent boundary.

A small subset of the subsystem's locks are involved in nested locking. The hierarchy is limited to the following:

higher position	device_lock, dev_number_lock
-	ip_lock
lower position	The rest of the subsystem's locks (no dependencies)

5.1 Device-IPC Layer

The **device** structure and the **io_req** structure are the central structures manipulated in this layer. I/O requests come to this layer via Mach IPC. Requests are handled as follows: the request's message is translated, via a hash table, to the **device** structure that represents the request's device. Once the device is identified, the I/O request is forwarded to the layers below via the **io_req** structure.

The **device** structure includes the following fields: **state**, **io_wait**, **open_count**, **port**, and **ref_count**. The first four fields are protected by the **device_lock**, though it must be noted that operations that manipulate the **port** field must hold the **device_lock** and take the associated **ip_lock** to manipulate the field safely. The **ref_count** field is protected by the **device-ref_lock**.

The remaining fields of the **device** structure are not protected by locks because they are set once at initialization time and are never changed. There is one exception; the field used to link the **device** structure to the **dev_number_hash_table** is protected by the system-wide **dev_number_lock**.

The use of the **device->ref_lock** as a fencing lock parallels similar mechanisms described in the other subsystems. It must be noted that in this case, a reference must be taken whenever an **ip_lock** or the **dev_number_lock** is held. For example, when a new element is being inserted into the device number hash table, the operation must hold take only the **dev_number_lock** because the new element is not known to other entities at this point. When an operation finds an element in the hash table, it must take the **dev_number_lock** first, then take the **device->ref_lock**.

The **io_req** structure does not require locking because the kernel assumes that it is manipulated by entities at different times and that these entities do not contend with one another. For example, the I/O request is created and initialized by the Device-IPC layer, and it passes through the char and block disciplines to the device dependent code. On the reverse path, **iodone( )** queues the I/O request for the **io_done_thread**, which executes the completion function specified by the I/O request. Ultimately, the completion function produces the MiG reply message.

There is one case in which agents may content for an **io_req** structure; this happens when one processor posts an **iodone( )** call and a thread is waiting for this event. The **ior_lock** was established to handle to prevent the race between the **thread_wakeup( )** and **assert_wait( )** calls.

5.2 Char, Block, and Network Devices

The discipline for a character device is in fact very similar to a standard UNIX tty line discipline. Similarly, the locking protocol is based on a simple lock per tty structure, the **t_lock** described above.

The discipline for a block device does not require locking because it uses MP safe primitives to manipulate the **io_req** structure (e.g.,, **io_wait( )**). The discipline also uses MP safe device dependent routines to perform the acutal I/O operations (e.g., *xxx*_**strategy( )**).

Network Devices

The network path is based on a network thread and ASTs. ASTs are used to activate the thread instead of relying on the **thread_block( )/thread_wakeup( )** sequence. The network data is managed via three queues:

net_queue_high
: Pending network messages that are of high priority. This queue is protected by the **net_queue_lock**.

net_queue_low

Pending network messages that are of low priority. This queue is protected by the **net_queue_lock**.

net_queue_free

Queue of empty buffers that are ready to be allocated. This queue is protected by the **net_queue_free_lock**.

The interrupt handler is in charge of deciding whether to place a given message on the high priority or the low priority queue. On most platforms, broadcast packets are placed on the low priority queue. In addition to protecting the low priority and high priority queues, the **net_queue_lock** also protects the boolean **network_thread_awake** that indicates whether or not the network thread is running

The following queues, on a per-network basis, require locks for consistency:

- Support for network packet filtering is provided by the **if_rcv_port_list** queue. This queue lists ports that network packets must be forwarded to. The queue is protected by the **(struct ifnet *)->if_rcv_port_list_lock**.

- In output, the traditional BSD TCP/IP mechanism for queueing **mbufs** has been converted into a mechanism that queues **io_req** structures. The **IF_ENQUEUE**, **IF_PREPEND**, and **IF_DEQUEUE** primitives have been made MP safe via the **(struct ifqueue *)->ifq_lock**.

6 Acknowledgements

This paper would not have been possible without the contributions of my colleagues who have expertise in specific subsystems. In particular I want to thank Bill Bryant for his help in editing the paper to clarify some of its major points.