A Mathematical Model of the Mach Kernel: Atomic Actions and Locks

William R. Bevier and Lawrence M. Smith

Technical Report 89 November 18, 1994

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: bevier@cli.com, lsmith@cli.com

Contents

1 Introduction

This report discusses the identification and use of Mach kernel atomic actions. Mach kernel calls may be programmed as compositions of these fine-grained steps. An atomic action has well-defined output properties, so a kernel call that is programmed as a sequence of atomic actions has predictable intermediate states. This is important for dealing with such issues as error recovery, pre-emption of kernel calls, security, kernel state testing, and reasoning about the concurrent execution of kernel calls. We also discuss an important related subject: the meaning of locks.

The identification of atomic actions is based on a model of a Mach kernel state in which relations on Mach entities are axiomatically introduced [BS93]. In this view, the *logical* form of a datum that is manipulated by the Mach kernel resembles a tuple in a relational database. The atomic action and locking interfaces present this logical view to the programmer, and hide the data structures that implement this view.

Section 2 of this report reviews the goals of our modeling work. Section 3 summarizes the main ideas of the Mach kernel state model. Section 4 discusses an interface to kernel atomic actions. Section 5 discusses locking primitives. Section 6 discusses some performance issues pertaining to atomic actions and locks.

2 Goals

The purpose of our work is to construct a model of the Mach kernel that captures the essence of its features and omits implementation details. Like conventional documentation, the model provides a conceptual framework for understanding the kernel. Unlike conventional documentation, the model is a self-contained description of the kernel. One need not refer to the implementation to understand a concept (but it is sometimes helpful). The model provides an independent and thorough statement of requirements. Such a model could become the definitive statement of requirements, and the source code could occupy a role as an efficient implementation of those requirements.

The interests of the authors go beyond documentation. We wish to apply tools to the analysis of the model. We use a theorem proving tool called Nqthm [BM88] (also known as the Boyer-Moore prover) to state and analyze the model. We have proved consistency of a part of the model with the theorem prover. We are investigating ways to model execution of kernel calls and prove theorems about these executions.

3 A Model of a Mach Kernel State

3.1 An Axiomatic Model

A kernel state consists of entities from several classes such as tasks, threads, ports, messages, pages, and memory objects. In [BS93] we identify relations in which entities may participate. For example, a *port right* is a relation involving a task and a port that characterizes a task's capability on a port. With axioms we place constraints on the relations that may hold in a kernel state — for instance, *at most one task may hold a receive right on a given port*. Our description of the entities, relations, and constraints provides a characterization of a legal Mach state.

To illustrate our approach, here is a brief presentation of some mathematical requirements on a legal Mach state. The requirements are expressed by identifying a collection of functions and predicates, and giving axioms about them. The description of each Mach concept involves a state variable σ. One thinks of a Mach property as holding in a given state.

We identify a recognizer predicate for elements of each of the Mach entity classes. For example, an entity x that satisfies the predicate $taskp(x, \sigma)$ is a task in state σ. (The p in the name *taskp* indicates that it is a predicate.) A thread is recognized by the predicate $threadp(x, \sigma)$. The entity classes are required to be mutually disjoint. Here is the axiom expressing the disjointness of *taskp* and *threadp*.

Axiom 1 $taskp(x, \sigma) \ \rightarrow\ \neg threadp(x, \sigma)$

The relation *task_thread* formalizes the notion of thread ownership. The expression $task\_thread(t, th, \sigma)$ holds in state σ if th is a thread in task t. For *task_thread* to hold, its arguments must be members of the right entity classes. This requirement is stated as follows.

Axiom 2 $\neg taskp(t, \sigma) \ \vee\ \neg threadp(th, \sigma) \ \rightarrow\ \neg task\_thread(t, th, \sigma)$

A thread may occur in at most one task. That is, the owning task of a thread is unique. This requirement is expressed by the following formula.

Axiom 3 $task\_thread(t_1, th, \sigma) \;\wedge\; task\_thread(t_2, th, \sigma) \;\rightarrow\; t_1 = t_2$

The expression $threads(t, \sigma)$ is the set of threads associated with task t. A thread th is an element of $threads(t, \sigma)$ if and only if is owned by t.

Axiom 4 $th \in threads(t, \sigma) \;\leftrightarrow\; task\_thread(t, th, \sigma)$

Some relations have non-entity attributes. For example, the port right relation has three attributes: a local name, a set of rights, and a reference count. The assertion $port\_right(t, p, n, R, i, \sigma)$ is understood to mean that task t holds rights to port p via local name n in state σ. The set $R \subset \{send, receive, send\_once\}$ identifies which rights are held, and i gives the reference count of the port right.

Currently, our mathematical description of Mach requirements includes about fifty relations on elements of the entity classes, and a large number of axioms on these relations. We believe that this is an accurate but incomplete set of requirements on a legal Mach state. More relations and constraints may be added after further investigation and public review.

A legal Mach kernel state can be defined from the complete set of axioms. For each axiom, construct a formula in which every free variable except σ is universally quantified, so that σ is the only parameter to the formula. Call this the *closure* of the axiom. For example, the closure of Axiom 1 is

$$\forall\, x\; (taskp(x, \sigma) \;\rightarrow\; \neg threadp(x, \sigma)).$$

A legal Mach state can be defined as the conjunction of the closures of the axioms.

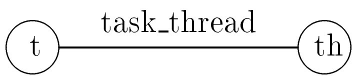
Definition 1 $legal\_state(\sigma) \;\equiv\; \forall\, x\; (taskp(x, \sigma) \;\rightarrow\; \neg threadp(x, \sigma)) \wedge \ldots$

We expect that Mach kernel atomic actions will be defined and used in such a way as to preserve legality. That is, an atomic action will produce a legal state from legal state.

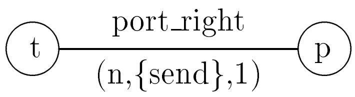
3.2 Other Views of the Model

A Mach kernel state can be visualized as a graph. An entity is a node. A relation is either a link between two nodes (possibly annotated with attributes), or is a line dangling from a single node in the case of a relation that involves a member of only one entity class. An example of the latter is the relation *dead_right*, which relates a task to a dead name in its name space.

If task t owns thread th, we imagine the following to be a part of the current state graph.



If task t holds a send right to port p, the following annotated link may occur in the state graph.



The graph representation of the axiomatic Mach kernel state model provides some useful intuition. Alternatively, one may think of the model in terms of a relational database[1]. Each Mach relation introduced in [BS93] corresponds to a relation in the database. For example, if task t owns thread th, the tuple [t,th] is an element of the TASK_THREAD relation. The requirement that a thread has at most one owning task suggests that the thread field of a TASK_THREAD tuple is the key of the relation.

An entity class is thought of as a relation consisting of singleton tuples. If t is a task, then the tuple [t] is in the TASK relation.

In the database literature, the term *consistency constraint* is used to denote what we have called a *legal state axiom*. Using our previous example, in a consistent state we don't allow tuples [t_1,th] and [t_2,th] in the TASK_THREAD relation if $t_1 \neq t_2$.

[1]See [KS86] for background on relational databases.

The relational database analogy provides an appealing logical view of a kernel state. The logical form of a datum that is manipulated by the Mach kernel is a tuple in some relation. As we shall see, the logical actions performed by the kernel can be viewed as tuple reads and writes. Locks are defined to hold sets of tuples fixed.

In the remainder of this report we are not careful to distinguish the axiomatic, graph, and database representations of the Mach kernel state model. We simultaneously think of an instance of a predicate $p(x, y, \sigma)$ as a link in a graph connecting nodes x and y, and as the tuple $[x,y]$ in relation P. We think of an entity recognizer as a node in a graph, or as a singleton tuple in a database relation that models the entity.

4 An Interface to Kernel Atomic Actions

4.1 Classification of Actions

The graph view of the Mach kernel state model suggests the following classification of Mach kernel atomic actions.

- Creation of a node (i.e., allocation of an entity).
- Destruction of an unconnected node (deallocation of an entity).
- Creation of a link (assertion of a relation)
- Destruction of a link (dis-assertion of a relation).
- Modification of a link attribute.

We find the database analogy to be useful for refining our intuition about classes of atomic actions. We may think of a program that implements a kernel call as a transaction on the kernel state. A transaction consists of a sequence of atomic actions. A fine-grained action on a database is the reading or writing of a single tuple, where writing includes insertion, deletion, or modification.

The following table identifies categories of Mach kernel atomic actions. The interpretation of the classes in terms of the axiomatic representation, the state graph representation, and the relational database representation are shown. Where there is not a conveniently described interpretation, we leave a table entry blank.

CLASS	Axiomatic	Graph	Database
ALLOC			allocate an unused relation key
INSERT	assert an instance of a relation	create a node or link	insert a tuple in a relation
DELETE	disassert a relation instance	remove a node or link	delete a tuple from a relation
MODIFY	disassert a relation instance, and assert a modification of it	modify a link attribute, or move a link	modify a tuple
READ			test the presence of a tuple, or read a tuple

Given this logical view of actions on a kernel state, we are faced with the necessity of defining interface functions within each class for the Mach kernel implementation. One can methodically examine all the relations given in [BS93] and identify the atomic actions in each class which can be derived from that relation. The method is not entirely algorithmic since within each class one must make various decisions that address practical issues: What are the desired input and outputs for an action? Which attributes of a relation may be modified? Are any of the entity fields of a relation modifiable? (This corresponds to atomically detaching a link from one node and attaching it to another.) In the next section we illustrate these decision points with examples.

4.2 Interfaces for Atomic Actions

Each atomic action is specified as a function from a set of inputs that includes the current state σ, to a set of outputs that includes a return code rc indicating the success or failure of the action. When the action changes state, a state variable σ' is among the outputs. We give the signature of an action by naming the inputs on the left side of an arrow, and the outputs on the right side. We use standard parameter names to denote types: t - task, th - thread, p - port, n - local name, i, j - natural numbers. Other types are introduced as needed.

Implementation Note. Comments on the current implementation of the Mach kernel are made in boxes. These can be ignored by the reader unfamiliar with or uninterested in the Mach implementation.

First, let us identify the atomic actions associated with an entity class. An entity class can be thought of as a relation consisting of singleton tuples. Since there are no attributes to this relation, we expect there to be no MODIFY actions on an entity relation. For the TASK relation we identify the following actions for each of the other classes.

Signature 1 *task_alloc:* $(\sigma) \rightarrow (t,\ rc,\ \sigma')$

Signature 2 *task_delete:* $(t,\ \sigma) \rightarrow (rc,\ \sigma')$

Signature 3 *task_read:* $(t,\ \sigma) \rightarrow (rc)$

A successful *task_alloc* finds an unused task identifier and inserts it as a task. One thinks of the resulting state graph as having a new, unconnected task node. *Task_delete* deallocates a task. *Task_read* tests for the presence of a task, i.e., it checks whether its argument is a task entity. *Task_alloc* combines the ALLOC and INSERT actions. Because there are no attributes to worry about, it makes sense to do both as one atomic step. We'll see later with port rights that it's natural to separate the steps of allocating an unused name and using it.

Implementation Note. Entity allocation in the Mach kernel is implemented by finding free space in some zone. Kernel consistency constraints require that at the time an entity is deallocated it participate in no relations with other entities. This is to avoid dangling pointers in other data structures. A reference count is used to keep track of the number of relations in which an entity participates. Deallocation of an entity's data structure is performed lazily only when its reference count reaches zero. The active bit of a data structure, when false, is used to prevent further insertions involving the entity implemented by that data structure.

Next we identify the atomic actions associated with the *task_thread* relation. Recall that this is a relation on a task and a thread, with the requirement that a thread may be associated with at most one task. This relation has no attributes to modify. Nor does it make sense, in this case, to permit modification of either of the entity fields. Therefore, we derive no MODIFY actions on this relation. Also, there are no ALLOC actions, since we rely on task and thread allocation to create the entities involved in this relation. Therefore we derive the following atomic actions.

Signature 4 *task_thread_insert:* $(t,\ th,\ \sigma) \rightarrow (rc,\ \sigma')$

Signature 5 *task_thread_delete:* $(t,\ th,\ \sigma) \rightarrow (rc,\ \sigma')$

Signature 6 *owning_task:* $(th,\ \sigma) \rightarrow (rc,\ t)$

Signature 7 *threads:* $(t,\ \sigma) \rightarrow (rc,\ th\_list)$

On success, *task_thread_insert* asserts the ownership of a thread by a task. *Task_thread_delete* dissolves this relationship. It just removes a link between the task and the thread, leaving the task and thread still in existence. The remaining two actions are from the READ class. The first, *owning_task* is the obvious atomic read on a *task_thread* tuple: given a thread (a key for the tuple) read its associated task. The other READ action is perhaps less obvious. This is the action that atomically reads all threads associated with a given task. Such actions that read multiple tuples are not necessary for every relation, but in the case of *task_thread* it is required to implement the kernel call `task_threads` that reports the names of all threads owned by a task.

Implementation Note. The assertion of a *task_thread* relation is accomplished in the Mach implementation by atomically setting a pointer in a thread to its owning task, and inserting the thread in a linked list in the owning task.

Finally, we identify atomic actions for the port right relation. Recall that a port right is a relation on a task, a port, a name, a set of rights, and a reference count in a given kernel state. Let the term $port\_right(t,\ p,\ n,\ R,\ i,\ \sigma)$ be an instance of a port right assertion. We consider n, R, i to be the attributes of this relation. The key is the pair $\langle t,\ n\rangle$ which means that the task and the name determine the other values. Since the name is a key, it is not a modifiable attribute of the relation.

Here are the candidate atomic actions on port rights. We have one ALLOC action: given a task, allocate an unused local name n. In the resulting state $\langle t, n\rangle$ can be used as a key in a port right relation.

Signature 8 *port_right_name_alloc:* $(t, \sigma) \rightarrow (rc, n, \sigma')$

The action that asserts a port right must be given initial values for all fields. The action that deletes a port right needs only a key for an argument.

Signature 9 *port_right_insert:* $(t, p, n, R, i, \sigma) \rightarrow (rc, \sigma')$

Signature 10 *port_right_delete:* $(t, n, \sigma) \rightarrow (rc, \sigma')$

There are decisions to be made concerning the MODIFY actions. Which attributes are modifiable? Are there groups of attributes that must be modified together? We have already justified the decision to make the name of a port right a non-modifiable parameter. In Mach, the set of rights and the reference count are independently modified. This suggests two atomic MODIFY actions.

Signature 11 *port_right_chg_rights:* $(t, n, R, \sigma) \rightarrow (rc, \sigma')$

Signature 12 *port_right_chg_refcount:* $(t, n, i, \sigma) \rightarrow (rc, \sigma')$

We face similar decisions with regard to READ actions. Given a key, for which fields do we wish to have a read action — all of them, some of them, grouped in any particular way? Secondly, do we need to read multiple tuples? For *task_thread* we identified the need to read all threads associated with a task. We've chosen to display the signature of only one READ action for port rights which is the action that returns all three of the attributes of a port right. It may prove to be more convenient to identify three separate interfaces.

Signature 13 *port_right_read:* $(t, n, \sigma) \rightarrow (rc, n, R, i)$

4.3 Summary

We have attempted to make the identification of Mach entities and relations given in [BS93] the only activity that requires “art”. Given this model, there are several obvious classes of atomic actions. Pragmatic decisions are required to derive interface functions for atomic actions within each class for each relation. The resulting interface gives a logical view of a kernel state that is easy to understand, to program and to formally model.

The identification of atomic actions is not algorithmic, but we have been able to impose some order. Three properties of a relation have an impact on the choice of atomic action.

- **Entities**. From the relational database point of view, the TASK relation has no priority over the TASK_THREAD relation. But, in fact, we make use of the knowledge that some relations model *entities* (i.e., nodes in a graph), and others model relationships between entities (links in a graph). This affects our decisions for atomic actions in many relations.

- **Keys**. A key is a collection of relation fields that determine a tuple. Some relations have more than one key. For example, the relation between a task and its self port is 1-1, and both are keys. We have not found a relation for the Mach kernel that has more than two keys.

- **Attributes**. Attributes are the non-entity fields of a tuple. The attributes of a relation may overlap with a key. For example, the name field of a port right relation is both an attribute and part of a key.

How does one identify the atomic actions that are associated with a relation R? One picks actions from each of the atomic action classes. The issues that determine choices within each class cannot be decided algorithmically from the relational model because they depend upon knowledge of which actions will be used, and how they will be used[2]. For each action class below, we indicate the questions that arise about a relation R in deciding which actions should be derived from R.

[2] Perhaps one can algorithmically generate the signatures for all possible atomic actions derived from a set of relations. This set is likely to be far larger than is useful.

ALLOC. An action from this class atomically finds an unused key for a relation.

- Does R have a key that requires allocation?

All of the entity relations have such a key. (Since there are no other attributes to an entity relation, it's reasonable to combine the key allocation and tuple insertion actions into one.) For other relations, we largely rely on entity allocation to supply the keys. That is, in any program, an entity that serves as the key of a relation will already have been allocated. There are some fields that still require allocation: local names, and virtual page addresses in an address space.

INSERT. An action in this class states a new fact, analogous to inserting a tuple in a database, or inserting a node or link in a graph.

- What is the signature of the INSERT action for R?

In some cases all fields of the tuple must be provided as arguments. In other cases, there are reasonable default initial values that may cause some fields to be omitted from the inputs. For example, 1 may be a reasonable default initial value for a port right reference count.

DELETE. An action in this class removes a fact (deletes a tuple).

- How many ways can an instance of R be deleted?

There is potentially one DELETE action for each key. The inputs for a DELETE action should be only a set of key fields.

MODIFY. An action in this class is the composition of a delete and an insert that is convenient to treat as atomic.

- Which are the modifiable attributes of R?
- Should any groups of such attributes be modified atomically?
- Which of the entity fields are modifiable?

The examples given in Section 4.2 elaborate on these issues. A modifiable entity field permits one to atomically detach a link from one node and attach it to another.

READ. A READ action senses the kernel state but does not change it. In database terminology, we restrict a read action to be the subsetting of a single relation. We do not support the construction of arbitrary views of multiple relations.

- For each of R's keys, what READ actions should be implemented?
- What aggregate READ actions should be implemented?

One expects there to be a read action with a key as input parameter. For relations with multiple non-key fields, there is a choice of signatures. Should there be one READ action for each non-key field, or should any of these fields be read in a group? Occasionally, as with *task_thread*, it is desirable to read all tuples that are associated with one or more non-key fields. These must be identified case by case.

The actions that result from applying this method are quite fine-grained. Many of them can be implemented with one line of C code. The interface to these simple ones can be defined with macros. Others, like insertion of a thread into a task, require modification of several locked entities. These can be defined as routines.

In the examples of the previous section, we have made some choices. Utility in coding kernel calls is the test for reasonableness of these choices. In our current work, we are applying this method to build an executable model of the kernel in the Nqthm logic [BM88]. This model will allow us to analyze both sequential and concurrent interpretations of kernel call implementations.

4.4 A Program that Uses Atomic Actions

In this section we display a pseudo code implementation of the kernel call `thread_create` in terms of atomic actions. This pseudo code corresponds to the routine `thread_create` in the kernel implementation. It does not describe the user callable interface. We assume that the parent task argument has been resolved to an actual task address, i.e., it is not the name of some port. For simplicity, we have left out some of the required behavior of this call. We do not show, for example, initialization of a thread's scheduling information and machine state.

The example shows a very literal application of an atomic action interface. As a result, some performance issues become apparent. For example, from an efficiency standpoint, there is an excessive setting of return codes. The purpose of this example is to illustrate the use of atomic actions. In Section 6 we discuss performance issues.

We've followed several conventions in the pseudo code. In the signature of `thread_create`, parameters to the left of the arrow are read-only inputs, parameters to the right can be written to and are visible to the caller. We've left out the state parameters σ and σ', which are useful for specifying the interface of an atomic action but are implicit in any real implementation. We have permitted outselves the use of an assignment statement that has multiple variables on the left hand side which we assume are performed in parallel. The return statement transmits a return code back to the caller.

We have omitted locks from this pseudo code. A set of atomic actions for locking and unlocking data structures must be added to the set of atomic actions (see Section 5). We have omitted from this implementation of `thread_create` the part which checks for the continued existence of the parent task.

```
thread_create (parent_task) → (child_thread)
        local: new_thread, port, procset;

        rc := task_read(parent_task);
        if rc ≠ SUCCESS
           child_thread := NULL_THREAD;
           return ( INVALID ARGUMENT);

        new_thread, rc := thread_alloc();
        if rc ≠ SUCCESS
           child_thread := NULL_THREAD;
           return ( RESOURCE_SHORTAGE);

        /* 1 */

        /*  Allocate the sself port.  */

        port, rc := port_alloc();
```

```
if rc ≠ SUCCESS
   child_thread := NULL_THREAD;
   return ( PANIC);

/* 2 */

rc := thread_sself_insert(new_thread,port);

/* 3 */

procset, rc := procset_task_read(parent_task);
if rc ≠ SUCCESS
   procset, rc := default_proc_set();
rc := procset_thread_insert(procset,new_thread);

/* 4 */

rc := task_thread_insert(parent_task,new_thread);

/* 5 */

/* Connect the thread to its self port. */

rc := thread_self_insert(new_thread,port);

/* 6 */

child_thread := new_thread;
return ( SUCCESS);
```

The main point to observe about this example is that one can write the `thread_create` kernel call in a natural way based on an atomic action interface. As a result of using this interface, the implementation has well-defined intermediate states. We have numbered some of these intermediate states. The assertions corresponding to each of these states can be summarized as follows.

1. A new thread exists.

2. A new port exists.

3. The port is inserted as the thread's sself port.

4. The thread is contained in the parent task's processor set, or in the default processor set.

5. The thread is associated with the parent task.

6. The new port is the thread's self port.

5 The Meaning of Locks

A lock may be used for two reasons: to implement an atomic modification to a state, and to hold some properties of a state fixed over a sequence of atomic steps. The former use requires a *write* lock, the latter use requires a *read* lock.

We seek an interface to locks at the same level of abstraction as the atomic transitions. The interface should have the following properties.

1. Each interface function locks a well-defined set of tuples.

2. Lock granularity provides reasonable concurrency.

3. Resolution of conflicting locks is efficient.

[EGLT76] proposes the notion of predicate locks, which allows one to lock arbitrary sets of tuples. This generality entails a potentially high cost in determining when two locking requests conflict. A less general interface that is more efficient in determining conflicts is desirable.

We apply locks to data structures that implement the abstract relations on Mach entities. To satisfy property 1, the set of abstract tuples held fixed by any data structure lock must be precisely defined. To satisfy 2, a data structure lock must hold a reasonably small number of tuples fixed. Data structure locks satisfy property 3 since one cannot lock an arbitrary set of tuples, but only those sets which correspond to some data structure.

In the following discussion we assume that Mach will continue to be implemented in an "entity-oriented" way.[3] That is, there will be a C data structure type corresponding to each entity class. Each of these data structures will contain (i.e., *implement*) a set of relations in which an entity may participate. This data structure organization is desirable because it provides efficient computation of an entity's properties, but it results in an implementation of the abstract relations on which the fields of a tuple are not directly represented as contiguous data in memory.

Let us assume for now that the C data structures corresponding to the abstract entities are the only lockable data structures in the implementation, and that there is exactly one lock per data structure. What set of tuples is held fixed by each lock in this implementation? To address this question, we must understand how each abstract relation is implemented by one or more entity structures. In the current Mach implementation a `thread` structure contains a pointer to its owning task, and a `task` structure contains a linked list of threads which it owns. Thus, the implementation of the *task_thread* relation involves two data structures, `task` and `thread`.

Detailed documentation on the implementation of each relation would be useful, but for the purposes of understanding the entity data structure locks, we need identify only the lockable structures that implement a relation, not the fields within a structure that are used. Given the above kind of analysis, we can construct a dictionary that associates with each relation a set of entity data structures that are involved in the implementation of that relation. We give some sample dictionary entries in Figure 1.

From this dictionary, one can build a cross reference that indicates, for each data structure, the tuples in various relations which this data structure implements. Figure 2 shows the cross reference for Figure 1. This cross reference defines the meaning of each data structure lock. According to this table, locking a `thread` structure holds fixed the set of all tuples, in any abstract relation, in which a given thread participates. Among the tuples implemented by a `thread` structure is the *task_thread* tuple containing the thread implemented by the pointer to that structure.

To acquire read access to a particular abstract tuple fixed, one applies a read lock to any one of the data structures that implement the tuple. To acquire write access to an abstract tuple, one holds a write lock on all of

[3]We refrain from using the heavily loaded term "object-oriented".

Relation	Implementing Data Structures
task_thread	`task` `thread`
port_right	`task` `port`
port_set	`task`
dead_right	`task`
task_self	`task` `port`
thread_self	`thread` `port`

Figure 1: Implementation of Relations

Data Structure	Implemented Relations
`task`	*task_thread* *task_self* *port_right* *port_set* *dead_right*
`port`	*port_right* *task_self* *thread_self*
`thread`	*task_thread* *thread_self*

Figure 2: Implementation Cross Reference

Data Structure	Implemented Relations
`task`	*task_thread* *task_self*
	port_right *port_set* *dead_right*
`port`	*port_right* *task_self* *thread_self*
`thread`	*task_thread* *thread_self*

Figure 3: Cross Reference Partition

the data structures that implement a tuple. There must be an agreed upon ordering for lock acquisition to avoid deadlock.

Having one lock per entity may or may not provide adequate granularity of locked tuples. The current Mach implementation includes more locks, but the authors are not aware of any tests that indicate whether or not the extra locks significantly increase concurrency.[4] If the extra locks do not significantly increase concurrency, then the simplicity of the one-lock-per-entity approach makes it a desirable implementation.

In case the extra locks are deemed desirable, one can modify the cross reference by partioning the table entries. Figure 3 shows a partition of the `task` entry, which indicates that port rights, port sets and dead rights are to be locked separately from the others. This partition exists in the current Mach implementation. These three relations are implemented by a task's `ipc_space` structure.

Partitioning raises the issues of a hierarchy of lockable structures. Does locking a task imply a lock on its IPC space, or is the space to be considered an independently locked set of tuples? If a hierarchy is intended, then the techniques for locking hierarchies outlined in [GLP75] can be used.

[4]This should be taken as a comment on the ignorance of the authors.

6 Performance Issues

In this section we discuss locks and return codes and how they impact performance.

6.1 Locks

A straightforward implementation of an atomic action begins with a request for one or more read or write locks, performance of some computation, followed by release of the locks. Since the atomic actions are fine-grained, this implementation approach may result in an unacceptable performance cost.

To minimize locking, the atomic actions may be implemented with the assumption that the caller has acquired the necessary locks. This will permit a caller to acquire locks that will span a number of atomic steps. While increasing performance, this approach runs the risk of coding errors. A program may erroneously fail to acquire a necessary lock.

A compromise exists. Implement each atomic action with locks. A calling program may acquire a lock that is redundantly requested by several atomic actions that it calls. These redundant requests can be eliminated at compile time by a pass through the code. In a typical case, the program will acquire all of the locks it needs for its atomic actions, and the compiler will not generate lock acquisition code for any atomic actions.

This lock-checker will help the programmer to avoid mistakes due to failure to acquire locks. When a program does not acquire a lock, an atomic action will protect itself at run time by acquiring and releasing a lock. The lock-checker can be engineered to report when locking code for an atomic action is generated, indicating a potential failure to acquire a lock by the caller.

6.2 Return Codes

Return codes are another source of inefficiency in the literal use of atomic actions. A return code makes each atomic action a total function. That is, it always returns some value: either success, or some flavor of failure.

In many cases, it can be predicted that an atomic action will not fail due to pre-conditions of a kernel call, or conditions tested on the code path to the call site of the atomic action. For example, when a TASK_THREAD tuple is inserted, and current locks guarantee the stability of this insertion, we can conclude that the READ action *owning_task* applied to the thread will succeed. There is no need to set or test a return code.

One rather tedious way to address this problem is to have two interfaces for each action: one that gives a return code, and one that does not. With a macro facility that performs compile time computation, a single interface can be defined with a flag parameter that lets the caller indicate whether the pre-condition on the action warrants a return code or not. As a side-effect, this would provide useful documentation throughout the code about intermediate assumptions.

7 Conclusion

We have presented a method for identifying atomic actions on a Mach kernel state in terms of which kernel calls can be programmed. The method is based on a relational model of the Mach kernel state. An atomic action has well-defined output properties, so a kernel call that is programmed as a sequence of atomic actions has predictable intermediate states. As a by-product of this analysis, we have discussed the abstract meaning of data structure locks.

It is possible to apply this method to a Mach kernel implementation in a partial way only on relations of interest. Other features of Mach can be dealt with in a traditional way. A thorough application of this method will require extension of the Mach kernel state model to include all relations of interest, and creation of a relation dictionary that defines how each relation is implemented in data structures.

The atomic action and locking interfaces present a logical view of a kernel state to the kernel call implementor, and hide the data structures that implement this view. This should result in cleaner, easier to maintain code. Additionally, these interfaces provide an abstraction of kernel behavior for which there is a tractable formal model.

References

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[BS93] William R. Bevier and Lawrence M. Smith. A mathematical model of the mach kernel: Entities and relations. Technical Report 88, Computational Logic, Inc., April 1993.

[EGLT76] K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, November 1976.

[GLP75] J.N. Gray, R.A. Lorie, and G.R. Putzolu. Granularity of locks in a shared data base. In *Proc. of the Intl. Conf. on Very Large Data Bases*, September 1975.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill, 1986.