# A Mathematical Model of the Mach Kernel

William R. Bevier and Lawrence M. Smith

Technical Report 102 December, 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.

**Abstract**

This report gives an overview of a mathematical specification for the Mach kernel, version 3.0. The specification describes a legal Mach kernel state and requirements on kernel requests. The specification admits implementations in which kernel requests execute concurrently. This is important since Mach is designed to run on multi-processor systems. The specification technique can be applied to other kernels, and to object-oriented systems in general.

# Contents

# 1 Introduction

Mach [Ras86] is an operating system kernel that has been under development for a number of years, primarily at Carnegie-Mellon University. It is not a fully functional operating system. It implements a few basic abstractions like *task*, *thread*, *message* and *port*. Usable operating systems are built on top of the Mach kernel in terms of these abstractions. We have written a detailed, partial specification for the functional behavior of the Mach kernel, version 3.0. The purpose of this report is to present our approach to specifying the kernel.

We have several goals in doing this work. The first is simply to provide mathematically precise documentation. As documentation, this report supplements existing sources [Loe91b], [Loe91a]. In them, Keith Loepere writes:

> Although it is a goal of the Mach kernel to minimize abstractions provided by the kernel, it is not a goal to be minimal in the semantics associated with those abstractions. As such, each of the abstractions provided has a rich set of semantics associated with it, and a complex set of interactions with other abstractions.
> — [Loe91b], pg 2.

This is an accurate characterization of the microkernel architecture. We believe that our mathematical formulation clarifies the essential features of Mach by precisely defining required behavior of the kernel interface, and ignoring implementation issues. Of course, by leaving out implementation issues we leave out much of what is interesting about Mach.

Our second goal is to begin the process of defining a contract between Mach users and implementors. It would be a benefit to the Mach community if an unambiguous statement of the required features of a Mach implementation were available. Programs which use only these features would be completely portable. This would make possible program portability at a high level of abstraction.

Our third goal is to enable the formal specification and proof of correctness of applications programs which run on Mach, and of programs and hardware which implement Mach.

The specification admits implementations in which kernel requests execute concurrently. This is important since Mach is designed to run on multiprocessor systems. Allowing for this concurrency introduces some complexity. The specification includes the following elements.

Legal Kernel State. We introduce the classes of entities that exist in the kernel, relations in which members of the entity classes may participate, and axioms about those relations. The legal state model suggests a collection of fine-grained atomic actions in terms of which the actions of kernel requests may be understood.

Specifications for Kernel Requests. A kernel request is modeled as a sequence of kernel states. A specification for a request is a predicate that defines a set of permitted behaviors. We use a temporal logic to write these specifications, primarily to give the partially ordered set of events which must take place during a kernel request.

Section 2 describes notational conventions. Section 3 introduces the legal state model with examples. Section 4 presents our approach to specifying kernel requests. Details of the specification can be found in two technical reports: [BS94a] and [BS94b].

# 2 Notation

## 2.1 Symbols

| | |
|---|---|
| $\mathbf{N}$ | the set of natural numbers |
| $=$ | equality |
| $\subseteq$ | subset |
| $\cap$ | set intersection |
| $\in$ | set membership |
| $\neg$ | negation |
| $\wedge$ | conjunction |
| $\vee$ | disjunction |
| $\rightarrow$ | implication |
| $\leftrightarrow$ | bi-implication |

| | |
|---|---|
| $\exists$ | existential quantification |
| $\forall$ | universal quantification |
| $+, -, *, \div$ | arithmetic operations on natural numbers |
| | E.g., $5 - 2 = 3$, but $2 - 5 = 0$. Also, $7 \div 2 = 3$ |
| $<, \leq, >, \geq$ | inequalities on natural numbers |
| $\langle a, b, c \rangle$ | a tuple |
| $\{a, b, c\}$ | a set |
| IDENT | a constant |
| `'ident` | a scalar constant |

## 2.2 Declarations

We specify Mach by introducing functions and predicates that represent Mach concepts, and by stating axioms about them. We introduce a new function symbol in a number of ways. A *defined* function is introduced as follows.

**Definition 2.1**
$\mathrm{f}(x,\ y) \equiv \mathrm{g}(x,\ y)$

Here, f is a new function symbol and g is an expression on f's arguments involving only previously defined functions.

When we intend only to partially specify a new function symbol, we introduce it with a sequence of declarations. The following form declares a new function symbol and the names of its formal parameters. This information determines the function's *arity*, that is, the number of its parameters.

**Function 2.2**
$\mathrm{f}(x,\ y)$

Subsequent axioms state assumptions about a function symbol, as in the following example. Sometimes we omit the printing of the function declaration, and let an axiom suffice to introduce a new function.

**Axiom 2.3**
$\mathrm{p}(x,\ y,\ z) \rightarrow (\mathrm{f}(x,\ y) = z)$

Some function symbols are predicates, i.e., functions whose range is the set of boolean values $\{true,\ false\}$. Certain predicates have particular prominence. A *relation* is a predicate on several arguments, the last of which is a state variable $s$. In the Mach specification, a relation holds on elements of one or more Mach entity classes, and optional additional parameters from other data types. We declare such a predicate in the following way.

**Relation 2.4**
$\mathrm{p}\,(x,\ \underline{y},\ s)$ WHERE
$\mathrm{q}\,(x,\ s) \wedge \mathrm{r}\,(y,\ s)$

This declaration introduces a new relation p along with the axiom

$$\mathrm{p}\,(x, y, s) = true \ \vee \ \mathrm{p}\,(x, y, s) = false.$$

The expression $\mathrm{q}\,(x,\ s) \wedge \mathrm{r}\,(y,\ s)$ can be thought of as a guard. The guard defines some necessary conditions for the relation p to hold; it introduces the axiom

$$\neg(\mathrm{q}\,(x, s) \wedge \mathrm{r}\,(y, s)) \rightarrow \neg \mathrm{p}\,(x, y, s).$$

While the guard can be an arbitrary predicate on the parameters to p, we typically write only elementary requirements. In our usage the guard looks like a *signature*, an expression which states the types of the parameters.

A set of parameters may be a key. As in database terminology, a key determines the other parameters of any instance of a relation. We indicate the members of a key with underlining. The state variable is a part of every key; we refrain from underlining it. For the above example, the following axiom is introduced for the key $y$.

$$p\,(x_1, y, s) \ \wedge \ p\,(x_2, y, s) \ \rightarrow \ x_1 = x_2$$

A relation may have more than one key. When there are two keys, we indicate the second key with overlining. The Mach specification currently has no relation with more than two keys.

Useful specification functions may be derived from a relation. In the relation p above, $y$ is a key. That is, in a given state, a single $x$ value may be p-related to multiple $y$ values. This suggests the following specification functions. The predicate *exists-x-related-to-y* holds if some $x$ is related to $y$ in state $s$. If so, the function *x-related-to-y* gives the unique $x$ related to $y$. The function *all-ys-related-to-x* is the set of $y$ values p-related to $x$ in state $s$.

**Definition 2.5**
exists-x-related-to-y $(y,\ s) \equiv \exists\ x$: p $(x,\ y,\ s)$

**Axiom 2.6**
exists-x-related-to-y $(y,\ s) \rightarrow$ (p (x-related-to-y $(y,\ s),\ y,\ s)$)

**Axiom 2.7**
q $(x,\ s) \rightarrow (y \in$ all-ys-related-to-x $(x,\ s) \leftrightarrow$ p $(x,\ y,\ s))$

# 3 A Legal Mach Kernel State

## 3.1 Mach Entities

The definition of each Mach concept involves a state variable $s$. One thinks of a Mach property as holding in a given state. A Mach kernel state contains entities from the following disjoint classes: tasks, threads, ports, messages, memories, pages, processors, processor sets, and devices.

A *task* is the unit of resource allocation. A task holds access to message ports and to memory. A task may contain one or more threads. A *thread* represents a flow of control within a task. One thinks of a thread as a program counter together with local cpu state. All threads share the resources allocated to the task in which they are contained. A *port* is container of messages. A task may hold the right to send a message to a port, and/or to receive a message from a port. A *message* is a unit of information which can be passed between two tasks. Messages can be used to pass data, and to pass rights to ports. An *abstract memory*, or just *memory*, is a unit of data. An abstract memory has roughly the semantics of a Unix file: it is a mapping from offsets to words. A task cannot access a memory directly—i.e., via a machine instruction. It can only directly access the contents of a page. A *page* is the unit of physical memory. A page is a fixed-size sequence of words. A task accesses a page via a virtual address. The primary purpose of a page is to hold a snapshot of some segment of an abstract memory. A *processor* is a hardware instruction interpreter. A *processor set* is a collection of processors. A *device* is one of a number of types of peripheral hardware.

We write *taskp* $(x,\ s)$ to say that $x$ is a task in state $s$. We call *taskp* a *recognizer* because it recognizes an element of one of the distinguished classes. The names of the other recognizers are *threadp*, *portp*, *messagep*,

*memoryp*, *pagep*, *procp*, *procsetp* and *devicep*. Here is the axiom that *taskp* may not recognize a member of any of the other entity classes. A analogous constraint applies to the other recognizers.

**Axiom 3.1**

$$\begin{aligned} & \text{taskp}\,(x,\ s) \\ \rightarrow \quad & \neg\ \text{threadp}\,(x,\ s) \\ \wedge\ & \neg\ \text{portp}\,(x,\ s) \\ \wedge\ & \neg\ \text{messagep}\,(x,\ s) \\ \wedge\ & \neg\ \text{memoryp}\,(x,\ s) \\ \wedge\ & \neg\ \text{pagep}\,(x,\ s) \\ \wedge\ & \neg\ \text{procp}\,(x,\ s) \\ \wedge\ & \neg\ \text{procsetp}\,(x,\ s) \\ \wedge\ & \neg\ \text{devicep}\,(x,\ s) \end{aligned}$$

In Mach, the kernel is viewed as a task. We introduce the constant KERNEL to represent the kernel task.

**Axiom 3.2**

$\text{taskp}\,(\text{KERNEL},\ s)$

## 3.2 Threads and Tasks

A task contains zero or more threads. The relation *task-thread-rel* associates a thread with a task. A thread belongs to at most one task.[1] The predicate *exists-owning-task* holds when a thread has an owning task, and *owning-task* identifies that task when such an assignment exists. The function *threads* is the set of threads associated with a task.

**Relation 3.3**

task-thread-rel $(t,\ \underline{th},\ s)$ WHERE
$\text{taskp}\,(t,\ s) \wedge \text{threadp}\,(th,\ s)$

**Definition 3.4**

exists-owning-task $(th,\ s) \equiv \exists\, t$: task-thread-rel $(t,\ th,\ s)$

[1] cf. [Loe91b], pg. 8

**Axiom 3.5**
exists-owning-task $(th,\ s) \rightarrow$ task-thread-rel (owning-task $(th,\ s),\ th,\ s)$

**Axiom 3.6**
taskp $(t,\ s) \rightarrow (th \in$ threads $(t,\ s) \leftrightarrow$ task-thread-rel $(t,\ th,\ s))$

As a result of these axioms, we can conclude that any element of the value of the function *threads* must be a thread.

**Theorem 3.7**
taskp $(t,\ s) \wedge th \in$ threads $(t,\ s) \rightarrow$ threadp $(th,\ s)$

## 3.3 Port Rights

Let $\mathcal{N}$ be a set. $\mathcal{N}$ is a set of names used to identify capabilities on ports. A task has access to a port only via a name in $\mathcal{N}$. We assume the existence of two distinguished names NULLNAME $\in \mathcal{N}$, and DEADNAME $\in \mathcal{N}$.

There are three access rights which a task can have on a port.[2] $\mathcal{R}$ is the set of port access rights.

**Definition 3.8**
$\mathcal{R} \equiv \{$`'send`, `'receive`, `'send-once`$\}$

A *port right* identifies a task's name for a port, and what by rights the task may access the port. In *port-right-rel*, $t$ is a task, $p$ is a port, $n$ is a name and $R$ is a non-empty subset of $\mathcal{R}$. The port right parameter $i$ can be thought of as representing the number of times a given port right has been granted to a task. This value is called the *reference count* of the port right. In any sequence of states, the value of $i$ is the number of times the right has been granted minus the number of times the right has been revoked. The reference count of a port right is a non-zero natural number less than the constant MAX-REFCOUNT.

A task and name determine the port in a port right relation, the set of rights held to it, and the reference count of the right. The predicate *port-right-namep* recognizes a task $t$ and name $n$ that represent a port right. The function *named-port* identifies the port to which task $t$ holds a right by name

[2] cf. [Loe91b], pg. 28

$n$. The function *port-rights* identifies the set of rights that task $t$ holds to a port by name $n$. The function *port-right-refcount* is the reference count of a port right.

**Relation 3.9**
port-right-rel $(\underline{t},\ p,\ \underline{n},\ R,\ i,\ s)$ WHERE
    taskp $(t,\ s)$
$\wedge$ portp $(p,\ s)$
$\wedge$ $(n \in \mathcal{N})$
$\wedge$ $(R \subseteq \mathcal{R})$
$\wedge$ $(0 < i <$ MAX-REFCOUNT$)$

**Definition 3.10**
port-right-namep $(t,\ n,\ s) \equiv \exists\ p,\ R,\ i$: port-right-rel $(t,\ p,\ n,\ R,\ i,\ s)$

**Axiom 3.11**
    port-right-namep $(t,\ n,\ s)$
$\rightarrow$ port-right-rel $(t,$ named-port $(t,\ n,\ s),\ n,$ port-rights $(t,\ n,\ s),$
                port-right-refcount $(t,\ n,\ s),\ s)$

Neither NULLNAME nor DEADNAME may serve as the name for a port right. The set of rights in a port right may not be empty.

**Axiom 3.12**
$(n =$ NULLNAME$) \vee (n =$ DEADNAME$) \rightarrow \neg$ port-right-rel $(t,\ p,\ n,\ R,\ i,\ s)$

**Axiom 3.13**
$\neg$ port-right-rel $(t,\ p,\ n,\ \emptyset,\ i,\ s)$

The reference count of a *receive* or *send-once* port right is exactly 1. A send right can have multiple references.

**Axiom 3.14**
port-right-rel $(t,\ p,\ n,\ \{$'receive$\},\ i,\ s) \rightarrow (i = 1)$

**Axiom 3.15**
port-right-rel $(t,\ p,\ n,\ \{$'send-once$\},\ i,\ s) \rightarrow (i = 1)$

**Axiom 3.16**
port-right-rel $(t,\ p,\ n,\ R,\ i,\ s) \wedge (R = \{\texttt{'send}, \texttt{'receive}\}) \rightarrow 2 \leq i$

The predicate *s-right* holds on a task and a name in state $s$ if and only if the name represents a send right in the task. The predicates *r-right* and *so-right* recognize names which represent receive and send-once rights, respectively, for a given task.

**Definition 3.17**
s-right $(t,\ n,\ s) \equiv$ port-right-namep $(t,\ n,\ s) \wedge$ `'send` $\in$ port-rights $(t,\ n,\ s)$

**Definition 3.18**
r-right $(t,\ n,\ s)$
$\equiv$ port-right-namep $(t,\ n,\ s) \wedge$ `'receive` $\in$ port-rights $(t,\ n,\ s)$

**Definition 3.19**
so-right $(t,\ n,\ s)$
$\equiv$ port-right-namep $(t,\ n,\ s) \wedge$ `'send-once` $\in$ port-rights $(t,\ n,\ s)$

A task has only one name for a send or receive right to a given port.[3] This is called *name coalescing*.

**Axiom 3.20**
s-right $(t,\ n_1,\ s)$
$\wedge$ s-right $(t,\ n_2,\ s)$
$\wedge$ (named-port $(t,\ n_1,\ s)$ = named-port $(t,\ n_2,\ s)$)
$\rightarrow (n_1 = n_2)$

**Axiom 3.21**
s-right $(t,\ n_1,\ s)$
$\wedge$ r-right $(t,\ n_2,\ s)$
$\wedge$ (named-port $(t,\ n_1,\ s)$ = named-port $(t,\ n_2,\ s)$)
$\rightarrow (n_1 = n_2)$

**Axiom 3.22**
r-right $(t,\ n_1,\ s)$
$\wedge$ r-right $(t,\ n_2,\ s)$
$\wedge$ (named-port $(t,\ n_1,\ s)$ = named-port $(t,\ n_2,\ s)$)
$\rightarrow (n_1 = n_2)$

[3] cf. [Loe91b], pg. 30

While send and receive rights to a port coalesce in a single name, a send-once right does not combine with others.[4] A task holding multiple send-once rights to a given port must hold them with distinct names.

**Axiom 3.23**
so-right $(t,\ n,\ s) \rightarrow \neg$ r-right $(t,\ n,\ s) \wedge \neg$ s-right $(t,\ n,\ s)$

At most one task can have a receive right on a port.[5]

**Axiom 3.24**
$$\begin{aligned} & \text{r-right}\,(t_1,\ n_1,\ s) \\ \wedge\ & \text{r-right}\,(t_2,\ n_2,\ s) \\ \wedge\ & (\text{named-port}\,(t_1,\ n_1,\ s) = \text{named-port}\,(t_2,\ n_2,\ s)) \\ \rightarrow\ & (t_1 = t_2) \end{aligned}$$

From the name coalescing property of receive rights, one can prove that $n_1 = n_2$ in the constraint above.

The identity of a port's receiver is a function of a port and a state $s$. We call this partial function *receiver*. The name by which a port's receive right is known to the receiver is given by *receiver-name*.

**Definition 3.25**
exists-receiver $(p,\ s) \equiv \exists\ t,\ n$: r-right $(t,\ n,\ s) \wedge$ (named-port $(t,\ n,\ s) = p$)

**Axiom 3.26**
$$\begin{aligned} & \text{exists-receiver}\,(p,\ s) \\ \rightarrow\ & \text{r-right}\,(\text{receiver}\,(p,\ s),\ \text{receiver-name}\,(p,\ s),\ s) \\ \wedge\ & (\text{named-port}\,(\text{receiver}\,(p,\ s),\ \text{receiver-name}\,(p,\ s),\ s) = p) \end{aligned}$$

## 3.4 Special Purpose Ports

The kernel assigns special meaning to some of its ports. Many of the special ports are used to make service requests on the kernel. The kernel holds the receive right on these ports. In other cases, the kernel holds a send right on a port, allowing it to asynchronously provide information to a user task.

[4] cf. [Loe91b], pg. 30
[5] cf. [Loe91b], pg. 25

There are other special ports to which the kernel has no rights. These port relations are maintained by the kernel in support of higher-level protocols.

A task may be associated with four special ports. A task self port (also called a task kernel port) identifies a task to the kernel and is used to perform actions in behalf of a task. The task exception port is used by the kernel to convey information about exceptions. The task bootstrap port is typically used for locating services. A task's sself port typically is identical to its self port. When task A's sself port differs from its self port, a debugging task holds a receive right on the sself port. The debugging task is said to *interpose* between the kernel and task A.

**Relation 3.27**
task-self-rel $(\underline{t},\ \overline{p},\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

**Relation 3.28**
task-eport-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

**Relation 3.29**
task-bport-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

**Relation 3.30**
task-sself-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

The special ports of a task are unique. Additionally, a task's self port is related to only one task. We introduce the function *task-self* to be the self port associated with a task. The other kinds of task special ports need not be related to only one task. The functions *task-eport*, *task-bport*, and *task-sself* have axioms analogous to *task-self*.

**Axiom 3.31**
task-self-rel $(t,\ p,\ s) \rightarrow$ (task-self $(t,\ s) = p$)

Only the kernel task may hold a receive right to a task self port.

**Axiom 3.32**
task-self-rel $(t,\ p,\ s) \wedge$ exists-receiver $(p,\ s) \rightarrow$ (receiver $(p,\ s) =$ KERNEL)

## 3.5 Comments on the Legal State Specification

This concludes examples of Mach kernel state specifications. We have described just enough to make intelligible an example specification for a kernel request in Section 4. Currently our Mach requirements include 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.

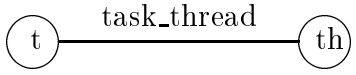
A legal Mach kernel state recognizer can be defined from the complete set of axioms. For each axiom, construct a formula in which every free variable except $s$ is universally quantified, so that $s$ is the only parameter to the formula. Call this the *closure* of the axiom. A legal Mach state can be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

**Definition 3.33**

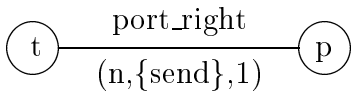
$$legal\_state(s) \equiv \forall x\ (taskp(x, s) \rightarrow \neg threadp(x, s)) \wedge \ldots$$

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

These classes identify the finest granularity step which have meaning in Mach. We believe that a Mach implementation which makes this interface explicit would be simpler to code and to understand. We use this level of granularity in writing temporal specifications for kernel requests.

### 3.6 Implementations of Mach Relations

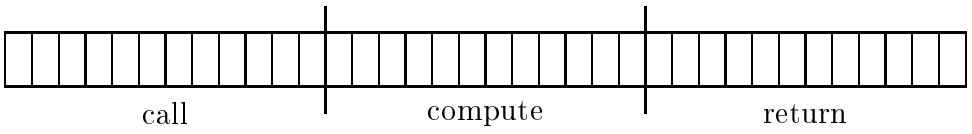
There is a straightforward implementation of each kernel relation in the C implementation of Mach. The entity recognizers are implemented by the addresses of C data structures. For example, $taskp(x, s)$ holds if one interprets $s$ as the memory occupied by the kernel, and $x$ as the address of a `task` structure as defined by the C code. The C structures which implement the other entity classes are `thread`, `ipc_port`, `ipc_kmsg` (a message), `vm_object` (an abstract memory), `vm_page`, `processor`, `processor_set`, and `device`.

Task-thread-rel $(t, th, s)$ is implemented by the `task` field of a `thread` structure. That is, when the `task` field of thread $th$ equals $t$, then thread ownership is established in state $s$. A `task` contains a header to a linked list of threads owned by the task. This suggests the implementation invariant that the `task` field of a thread $th$ must point to the task in whose thread list $th$ is linked.

## 4 Kernel Request Specifications

### 4.1 Kernel Behaviors

We have specified a subset of the kernel requests described in the Mach kernel interface manual [Loe91a]. The implementation of a kernel request is modeled




as a *behavior*, which is a sequence $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_k} s_k$ where each $s_i$ is a state and each $a_i$ is an *agent* (see below). Each state in a behavior is produced from its immediate predecessor by an atomic action. This execution model is compatible with state-based formalisms for concurrent computation like TLA [Lam91] and Unity [CM88].

Associated with each state in a kernel behavior is an agent identifier. This identifier names the entity responsible for creating the corresponding state from the previous one in the behavior.[6] We think of the agent as the entity in whose behalf a kernel behavior takes place, so that the steps involved in a single thread of kernel behavior, and only those steps, are labeled with the same agent.

From the point of view of a calling task we may divide a kernel behavior into three phases: asynchronous call, internal computation, and asynchronous return.

Concurrent steps in behalf of other agents may occur in each of these three phases. The steps associated with two distinct interleaved kernel behaviors have different agents. Some kernel services are implemented with both synchronous and asynchronous interfaces. In a synchronous interface, there is no concurrency in the call and return phases. A few kernel services have only a synchronous interface. We ignore the exceptions, and specify only the general case in which concurrency may occur in all phases. The resulting specifications apply to the synchronous interfaces as well, but are somewhat weaker than necessary.

[6]By *entity*, we do not necessarily mean a kernel entity, but a more abstract notion of the name of the service invoker. If one must be concrete, one can think of the agent as a stack of thread identifiers constructed by remote procedure calls. The topmost element of the stack is the identifier of the thread executing in behalf of the next thread on the stack, and so on.

There is very little that we can say about a kernel request in the state at which the return phase terminates. Conditions which hold in the initial state may be modified by interfering concurrent behaviors. The kernel entities that one expects a kernel service to operate on may disappear in each of the three phases. In the compute phase, the kernel has most control since it may lock the data structure that represents an entity, thus preventing (with the cooperation of other kernel threads) modifications to its properties.

Because of the possibility of interfering concurrent kernel computations, specifying a kernel request by stating pre-conditions and post-conditions is inadequate. Such an approach prohibits many computations that we want to consider legal. Instead, we use temporal logic to write predicates that describe legal implementations of kernel requests.

## 4.2 Temporal Logic

A behavior is specified in terms of two kinds of properties: *safety* and *liveness*. A safety property is one that is true of all states in a behavior. A liveness property is one that eventually holds. These notions are expressed formally in [AS85]. In the case of Mach, kernel behaviors do not cooperate to achieve some result, they merely interfere with one another. We therefore expect that a liveness specification for a kernel behavior is a formula that mentions at most one agent.

We use a temporal logic to express behavior specifications. The syntax of a temporal logic includes application of temporal operators to state predicates, and to other temporal predicates. The meaning of a temporal operator can be defined in terms of quantification over states in a behavior. For example, the expression $\Diamond p$, pronounced *eventually* $p$, is a predicate on a behavior that says there exists a state in which $p$ holds.

In the remainder of this section we informally summarize our temporal logic. This logic is fairly standard. It is similar, for example, to that found in [Pnu85]. The only novelty is that we use a notation which makes it easy to specify the actions of particular agents.

### State Terms

A *state term* is a term that has meaning in a kernel state. We build up state terms from constants, variables and function application. Constants and

variables take values from underlying domains which include integers, sets, symbols and sequences. Functions include equality ($=$), boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$), bounded quantification ($\forall$, $\exists$), set operations (e.g., $\subset$, $\cap$, $\cup$), and integer operations (e.g., $+$, -, $\times$, $\leq$).

Most important for specifying the Mach kernel, the functions introduced in the legal state model [BS94a] are included in the set of state functions. Most of the functions introduced in [BS94a] are state functions. For example, *threads* (see pg. 6 of this report) is a function on a state that returns the set of threads associated with a given task. A *state predicate* is a boolean-valued state function.

Within our temporal logic we omit the state variable from a state term. The term $taskp\,(x)$ is a state predicate that can be applied to any state in a behavior.

### Primitive Temporal Operators

We summarize our temporal logic with examples of the use of temporal operators. A *temporal term* is a term in the logic whose main operator is one of those discussed below. The unstated argument of a temporal term is a behavior of the form $\xrightarrow{\alpha_0} s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_k} s_k$. An $\alpha$-*state* is the resulting state of a step by agent $\alpha$.

**Now.** Behavior $\sigma$ satisfies $p$ iff $p$ is a state predicate that holds in $\sigma$'s first state. $\sigma$ satisfies $p[\alpha]$ iff $p$ holds in $\sigma$'s first state and the first state is an $\alpha$-state. The empty behavior satisfies neither $p$ nor $p[\alpha]$.

**Next.** If $p$ is a state predicate, behavior $\sigma$ satisfies $\odot p$ iff $\sigma$ has at least two states and $p$ holds in $\sigma$'s second state.

**Steady.** Behavior $\sigma$ satisfies $|x$ (the operator is pronounced *steady*) if $x$ is a state term whose value does not change in the first two states. A behavior whose length is less than two satisfies $|x$. Behavior $\sigma$ satisfies $|x[\alpha]$ if $x$'s value is steady and the agent of the step into the second state is $\alpha$.

**Eventually.** If $p$ is a state predicate, behavior $\sigma$ satisfies $\Diamond p$ iff there exists a state in $\sigma$ in which $p$ holds.

**Always.** If $p$ is a state predicate, behavior $\sigma$ satisfies $\Box p$ iff $p$ holds in every state of $\sigma$.

**Sequential Composition.** If $p$ and $q$ are temporal terms, behavior $\sigma$ satisfies $p \ ; \ q$ if some initial subsequence of $\sigma$ satisfies $p$, and the remainder of $\sigma$ satisfies $q$.

### Defined Temporal Operators

**Assertion.** A state predicate $p$ is *asserted* by agent $\alpha$ in a behavior, written $\uparrow p[\alpha]$, iff $\neg p$ holds now, $p$ holds in the next state, and the next state is an $\alpha$-state. We write $\Diamond\uparrow p[\alpha]$ to say that $\alpha$ eventually asserts $p$. $\downarrow p[\alpha]$ is defined to be $\uparrow (\neg p)[\alpha]$.

$$\uparrow p[\alpha] \equiv \neg \ p \ \wedge \ (\odot(p[\alpha]))$$

**Interference.** State term $x$ is observes *interference* with respect to agent $\alpha$ if the step into its second state is a $\beta$-step for some $\beta \neq \alpha$, and $x$ is not steady. This is written $\dagger x[\alpha]$. We write $\Diamond\dagger x[\alpha]$ to say that interference eventually occurs on $x$.[7]

$$\dagger x[\alpha] \equiv \neg \ (\odot(\mathbf{t}[\alpha])) \wedge \neg \ |x$$

**Protection.** A state predicate $p$ is *protected* in a computation, written $\|p$, iff $p$ is steady when $p$ holds in the first state. $p$ is *$\alpha$-protected*, written $\|p[\alpha]$, if it is protected when the first step of the computation is an $\alpha$-step.

$$\|p \equiv p \ \rightarrow \ |p$$

$$\|p[\alpha] \equiv (\odot(\mathbf{t}[\alpha])) \rightarrow \|p$$

These notions are most useful in combination with $\Box$. A computation that satisfies $\Box\|p$ is one in which $p$ stays true once it becomes true. We say that $p$ is *stable*. A computation that satisfies $\Box\|p[\alpha]$ is one in which agent $\alpha$ does not dis-assert $p$ if $p$ ever becomes true.

[7] The notation $\odot(\mathbf{t}[\alpha])$ says that $True$ holds in the second state and that this state is an $\alpha$-state.

### Operator Precedence

The order of operator precedence is $\neg$ followed by the set $\uparrow$, $\downarrow$, $|$ and $\|$, followed by $\Diamond$ and $\Box$, followed by the logical connectives $\wedge$ and $\vee$. Thus, $\|\neg \mathrm{p}\,(x)[\alpha]$ is the same as $\|\,(\neg p(x))[\alpha]$. $\Diamond p \wedge \Diamond q$ is the same as $(\Diamond p) \wedge (\Diamond q)$.

### Common Patterns

The formula $\Diamond p \;;\Diamond q$ recognizes a behavior in which $p$ eventually holds, and $q$ holds subsequently. This is the basic pattern for specifying an order to events. When we want to specify that all of $q1$, $q2$, $q3$ happen after $p$, but not give an order, we write $\Diamond p \;;\Diamond q1 \wedge \Diamond q2 \wedge \Diamond q3$

### Named Formulas

A temporal formula may be named. One understands a reference to a named formula as a copy of the name's definition. For example, the two definitions of FORMULA1 below are identical.

**Definition 4.1**
FORMULA1 $\equiv \mathrm{q}\,(x)[\alpha] \wedge$ FORMULA2

**Definition 4.2**
FORMULA2 $\equiv \Diamond\uparrow \mathrm{r}\,(x)[\alpha]$

**Definition 4.3**
FORMULA1 $\equiv \mathrm{q}\,(x)[\alpha] \wedge (\Diamond\uparrow \mathrm{r}\,(x)[\alpha])$

We permit names to take arguments. An argument represents a term that may be substituted for at "formula construction time". The two definitions of FORMULA3 below are identical.

**Definition 4.4**
FORMULA3 $\equiv (\Diamond \mathrm{q}\,(x)[\alpha]) \wedge$ formula4 $(\mathrm{f}\,(x))$

**Definition 4.5**
formula4 $(x) \equiv \Diamond\uparrow \mathrm{r}\,(x)[\alpha]$

**Definition 4.6**
FORMULA3 $\equiv (\Diamond \mathrm{q}\,(x)[\alpha]) \wedge (\Diamond\uparrow \mathrm{r}\,(\mathrm{f}\,(x))[\alpha])$

## 4.3 Specification Example

A Mach kernel request is specified by one or more temporal predicates. A behavior that does not satisfy a service's specification is not an implementation of the service. In this section, we discuss some patterns and conventions in the kernel specifications.

In the Mach 3.0 implementation, an interface parameter that represents an entity (e.g., the parent task in the `task_create` example above), is intended to be the caller's name for a port that represents the entity. This port name is resolved to an entity pointer early in the computation. We assume that this resolution has been performed. That is, the free variables denote entities, not the names of ports that represent entities.

The interface to a kernel service is specified in [Loe91a] by a C language routine header. For example, the interface to the service `task_create` is described as follows.

kern_return_t **task_create** (task_t *parent_task*,
boolean_t *inherit_memory*,
task_t **child_task*)

The arguments are a parent task from whom the newly created task inherits certain resources, an *inherit memory* flag which governs whether or not the new task inherits the parent's address space, and the returned child task pointer. An implicit out parameter is a return code `rc` specified by the function type `kern_return_t`.

We specify `task_create` with a temporal predicate Task-Createp that recognizes an acceptable `task_create` behavior. An interface parameter is represented by a variable in the predicate. There is an additional variable in each specification: $\alpha$, the agent of the request.

We follow the convention of printing the free variables in a **bold** font. Other variables, introduced by quantification, are not so printed. The function symbols of state terms are printed lower case. The function symbols of temporal predicates are printed with partial capitalization.

## PARAMETERS

$\mathbf{t_1}$. The parent task.

**inh-flg**. If **inh-flg** is *True*, the child's address space is inherited from the parent according to inheritance values at each of the parent's allocated virtual page addresses. Otherwise, the child's address space is empty.

$\mathbf{t_2}$. [out] The child task.

## OUTCOMES

We specify three possible outcomes: *success*, *invalid argument*, or *resource shortage*.

Task-Createp
$\equiv$ Task-Create-Success
$\vee$ Task-Create-Invalid-Arg
$\vee$ Task-Create-Resource-Shortage

## SPECIFICATION

On a successful outcome, $\mathbf{t_1}$ is found to be a task, and the child task is created and initialized.[8]

Task-Create-Success
$\equiv$ $\Diamond$taskp $(\mathbf{t_1})[\alpha]$
; $\Diamond\uparrow$taskp $(\mathbf{t_2})[\alpha]$
; Task-Initialized
; $\Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]$

Initialization includes creation of several special ports, and inheritance of the parent's address space and processor set.

Task-Initialized
$\equiv$ Task-Self-Created
$\wedge$ Task-Bport-Initialized
$\wedge$ Task-Eport-Initialized
$\wedge$ (**inh-flg**[`'alpha`] $\rightarrow$ Task-Memory-Inherited)
$\wedge$ ($(\neg$ **inh-flg**$)$[`'alpha`] $\rightarrow$ Task-Memory-Not-Inherited)
$\wedge$ Task-Procset-Inherited

[8]The form $\Diamond$taskp $(\mathbf{t_1})[\alpha]$ requires that an *$\alpha$-state* is reached in which the parameter $\mathbf{t_1}$ is a task. Intuitively, this requires a check to this effect. The form $\Diamond\uparrow$taskp $(\mathbf{t_2})[\alpha]$ requires that $\mathbf{t_2}$ be created in an *$\alpha$-step*.

A port is created and is made the child's self and sself ports.

Task-Self-Created
$\equiv \exists\, p \in$ ALL-ENTITIES:
$(\quad \Diamond\uparrow \text{portp}\,(p)[\alpha]$
$;\quad \Diamond\uparrow \text{task-self-rel}\,(\mathbf{t_2},\, p)[\alpha]$
$\wedge\ \Diamond\uparrow \text{task-sself-rel}\,(\mathbf{t_2},\, p)[\alpha])$

The child either inherits its parent's bootstrap port, or the parent is found to have no bootstrap port, and so the child is assigned none. An analogous specification holds for the child's exception port.

Task-Bport-Initialized
$\equiv \quad \exists\, p \in$ ALL-ENTITIES:
$(\quad \Diamond \text{task-bport-rel}\,(\mathbf{t_1},\, p)[\alpha]$
$;\ \Diamond\uparrow \text{task-bport-rel}\,(\mathbf{t_2},\, p)[\alpha])$
$\vee\ (\quad \Diamond(\neg\ \text{exists-task-bport}\,(\mathbf{t_1}))[\alpha]$
$;\ \Diamond(\neg\ \text{exists-task-bport}\,(\mathbf{t_2}))[\alpha])$

Task-Eport-Initialized
$\equiv \quad \exists\, p \in$ ALL-ENTITIES:
$(\quad \Diamond \text{task-eport-rel}\,(\mathbf{t_1},\, p)[\alpha]$
$;\ \Diamond\uparrow \text{task-eport-rel}\,(\mathbf{t_2},\, p)[\alpha])$
$\vee\ (\quad \Diamond(\neg\ \text{exists-task-eport}\,(\mathbf{t_1}))[\alpha]$
$;\ \Diamond(\neg\ \text{exists-task-eport}\,(\mathbf{t_2}))[\alpha])$

If **inh-flg** =*true*, each allocated virtual page address (vpa) in the parent task is allocated to the child according to the inheritance value associated with the parent's vpa as follows.

None. This vpa is not allocated in the child. n

Share. The memory mapped into the parent's address space is mapped into the child's at the same virtual address.

Copy. A copy of the memory mapped into the parent's address space is mapped into the child's address space. A new, temporary memory is created.

If a vpa is not allocated in the parent, then it is not allocated in the child.

Task-Memory-Inherited
$\equiv \forall\ 0 \leq \mathit{vpa} < \text{ADDRESS-SPACE-LIMIT}:$
$\quad (\quad \text{page-aligned}\,(\mathit{vpa})[\alpha]$
$\quad \rightarrow \quad (\quad \Diamond(\neg\ \text{allocated}\,(\mathbf{t_1},\ \mathit{vpa}))[\alpha]$
$\quad\quad\quad ;\ \ \Diamond(\neg\ \text{allocated}\,(\mathbf{t_2},\ \mathit{vpa}))[\alpha])$
$\quad\quad \vee\ \text{Task-Memory-None}\,(\mathit{vpa})$
$\quad\quad \vee\ \text{Task-Memory-Share}\,(\mathit{vpa})$
$\quad\quad \vee\ \text{Task-Memory-Copy}\,(\mathit{vpa}))$

We omit the definitions of Task-Memory-None, Task-Memory-Share and Task-Memory-Copy since they involve relations which formalize a task's address space that we have not presented in this report.

If **inh-flg** =*false*, then no virtual addresses are allocated in the child.

Task-Memory-Not-Inherited
$\equiv \forall\ 0 \leq \mathit{va} < \text{ADDRESS-SPACE-LIMIT}:\ (\Diamond(\neg\ \text{allocated}\,(\mathbf{t_2},\ \mathit{va}))[\alpha])$

The child is assigned to the parent's processor set if the parent has one, otherwise the child is assigned to the default processor set.

Task-Procset-Inherited
$\equiv \quad \exists\ \mathit{procset} \in \text{ALL-ENTITIES}:$
$\quad\quad (\quad \Diamond\text{procset-task-rel}\,(\mathit{procset},\ \mathbf{t_1})[\alpha]$
$\quad\quad\ ;\ \ \Diamond\uparrow\text{procset-task-rel}\,(\mathit{procset},\ \mathbf{t_2})[\alpha])$
$\vee\ (\quad \Diamond(\neg\ \text{exists-task-assigned-procset}\,(\mathbf{t_1}))[\alpha]$
$\quad\ ;\ \exists\ \mathit{procset} \in \text{ALL-ENTITIES}:$
$\quad\quad\quad (\quad \Diamond\text{default-procset-rel}\,(\mathit{procset})[\alpha]$
$\quad\quad\quad\ ;\ \ \Diamond\uparrow\text{procset-task-rel}\,(\mathit{procset},\ \mathbf{t_2})[\alpha]))$

An *invalid argument* outcome occurs if $\mathbf{t_1}$ is discovered not to be a task.

Task-Create-Invalid-Arg
$\equiv \Diamond(\neg\ \text{taskp}\,(\mathbf{t_1}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-arg})[\alpha]$

Task-Create-Resource-Shortage
$\equiv \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha]$

## 4.4 Comments on Kernel Request Specifications

We have specified approximately 40 kernel requests with this temporal logic, including the complex `mach_msg_send` and `mach_msg_receive`. The specifications give primarily liveness requirements — actions that a request must take — and the conditions under which they occur. These properties correspond to the documented behavior for each kernel request in the interface manual [Loe91a]. The length of the text required to present these formal requirements does not greatly exceed the English-only text.

We have chosen to omit safety properties from the kernel request specifications. Such properties capture what a kernel request may not do. For example, `task_create` should not change the port rights of any task other than the created child task.

$$
\begin{aligned}
&\text{NO-CHANGED-PORT-RIGHTS} \\
\equiv\ &\forall\, t \in \text{ALL-TASKS},\, p \in \text{ALL-PORTS},\, n \in \mathcal{N},\, r \in \mathcal{R}, \\
&\quad 0 \leq i < \text{REF-COUNT-LIMIT}: \\
&\quad (t \neq \mathbf{t_2} \rightarrow (\Box|\text{port-right-rel}\,(t,\, n,\, p,\, r,\, i)[\alpha]))
\end{aligned}
$$

A complete specification should include such requirements. Lamport argues that these are best described with an abstract program [Lam89]. We've excluded them in the interest of economy.

The liveness specifications capture much of the information contained in the English language documentation. They formally describe a partial ordering of events and conditions that must occur during a kernel computation. In the absence of interference (something that the kernel does not guarantee) the kernel state properties asserted by an agent hold in the final state of a computation. More precisely, one can conclude that property $p$ holds in the final state of a computation if $p$ holds eventually, $p$ is always protected by agent $\alpha$, and $\alpha$ observes no interference in property $p$. These conditions can be expressed formally as follows.

$$\Diamond p \wedge \Box||p[\alpha] \wedge \neg\, (\Diamond\dagger p[\alpha])$$

For a given kernel computation and property $p$, the first of these conjuncts follows from the liveness specifications that we give. The second follows from unstated safety specifications - e.g., that once $p$ is asserted, $\alpha$ does not disassert it. The last conjunct relies on reasoning about the environment in which the computation occurs.

# 5 Conclusion

The Mach 3.0 kernel specification is derived from an existing implementation. We constructed it by inspecting the documentation and source code for a number of Mach 3.0 releases out of CMU. The specification includes two components: a formalization of a legal kernel state, and requirements on kernel requests. The former is an abstract description of the architecture of the system; it says what may exist in a kernel state. The latter describes the dynamic behavior of the system.

The legal state specification is based on a mathematical foundation of first-order logic, set theory and arithmetic. The system architecture is defined by identifying classes of entities, and the relations in which they may occur. This approach seems suitable for many object-oriented systems. The kernel request specification requires an additional mathematical foundation of temporal logic. We have chosen to use a temporal logic to write a specification that will admit concurrent implementations. The temporal logic we have presented is a standard one. Similar notions can be found in [MP81], [Pnu85], and [Lam91]. The only novelty in our temporal notation is the prominence of agents.

We felt it important to express the specification in as simple a logic as possible. It does not depend on any particular existing specification system or tool. However, we believe that this specification could easily be cast into any number of them. We have gone to the effort of interpreting this specification in the logic of Nqthm, the Boyer-Moore theorem prover [BM88]. We have used Nqthm to prove the consistency of the legal state specification, and to check the well-formedness of our kernel request specifications. Secure Computing Corporation has transcribed the legal state requirements into the Z notation [Spi89], and extended them for the purpose of specifying a distributed, trusted version of Mach kernel [FMS93].

The legal state specification suggests a fine-grained set of atomic kernel steps, namely, steps which assert or disassert instances of the primitive Mach state relations described in [BS94a]. We believe that this is the finest granularity of step that makes sense for Mach. [BS93] discusses atomicity and locking issues in more detail.

In future work, we will apply this specification approach to functional and security requirements on servers running above the kernel. We will address the problem of modeling kernel and server implementations so that we can

prove that the specifications are satisfied.

We are currently working in collaboration with the Open Software Foundation Research Institute to modify this specification so that it applies to their Mach++ design [Fou93]. The main goal of this project is to implement tests for compliance with the specification. We hope that one result of this project is that a mathematical kernel specification will become a standard part of system documentation, and will be updated in concert with the implementation and tests as the design evolves.

# References

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[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: Atomic actions and locks. Technical Report 89, Computational Logic, Inc., April 1993.

[BS94a] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Entities and relations. Technical Report 88, Computational Logic, Inc., December 1994.

[BS94b] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Kernel requests. Technical Report 53, Computational Logic, Inc., December 1994.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design, a Foundation*. Addison Wesley, 1988.

[FMS93] Todd Fine, Carol Muehrcke, and Edward A. Schneider. Formal Top Level Specification for Distributed Trusted Mach. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1993. DTMach CDRL A012.

[Fou93] Open Software Foundation. Trusted mach kernel executive summary. Technical Report 0034-93A, Open Software Foundation, November 1993.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *CACM*, 32-1, 1989.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, December 1991.

[Loe91a] Keith Loepere. Mach 3 kernel interface. Technical report, Open Software Foundation, May 1991.

[Loe91b] Keith Loepere. Mach 3 kernel principles. Technical report, Open Software Foundation, March 1991.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

[Pnu85] A. Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.* Springer-Verlag, New York, 1985.

[Ras86] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8), August 1986.

[Spi89] J.M. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall, 1989.