A Mathematical Model of the Mach Kernel: Entities and Relations

William R. Bevier and Lawrence M. Smith

Technical Report 88 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.

Contents

1 Introduction

Mach [?] 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.

This is the first in a series of reports that give a mathematical model of the functional behavior of the Mach kernel version 3.0. We have several goals in doing this work. The first is simply to provide mathematically precise documentation. As documentation, this report supplements existing sources [?], [?]. 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.
> — [?], 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.

A benefit of having a mathematically precise contract is that it can be used to resolve disputes about what is compatible with Mach. Is a new feature purely an extension of Mach — or does it modify existing functionality? Such questions can be addressed by proof, not only by testing. This contrasts with programming language based efforts at standardization like [?].

Our third and final goal is to begin research on the formal specification and proof of correctness of applications programs which run on Mach, and of programs and hardware which implement Mach. Ultimately, we would like to have a Mach kernel specification in a form which can be used by a mechanical proof checker. This report provides a first step in that direction.

This report describes the primitive entities and properties of those entities in a Mach kernel state. It states axioms about a "legal" Mach state. Subsequent reports will deal with transitions on a Mach state, ultimately leading to formal specifications for kernel calls.

This report can be viewed as an introduction to Mach, but it assumes at least a familiarity with the overall design goals of the Mach project. We have not attempted to provide complete explanations for every concept introduced. We rely heavily on the existing literature, and make frequent references to [?] for corroboration.

This report is derived from an annotated "script" of events submitted to Nqthm, the Boyer-Moore theorem prover [?]. A script is a file containing function definitions, axioms and theorems in the Nqthm logic. We have arranged for Nqthm to process this script as follows. It checks that all applications of function symbols are syntactically correct, all suggested theorems are in fact true, and all definitions are well-formed. We have used the theorem prover to demonstrate the consistency of the axioms introduced in this report (see Section 10). This report can be thought of as a guide to the Nqthm script, which contains all of the details of our formalization of a Mach kernel state. We have suppressed some of the details in this report for the sake of readability.

Implementation Note. To help motivate and explain our formalization, we refer to the existing Mach implementation in C. These implementation notes are set off from the rest of the text, and may be ignored by the reader who is unfamiliar with, or uninterested in, the Mach implementation.

2 Notation

2.1 Symbols

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 Operator Precedence

We use infix notation for many mathematical expressions. When parentheses are omitted, this can result in ambiguities. Does $a * b + c$ mean $(a * b) + c$ or $a * (b + c)$? To resolve such ambiguities, we resort to a convention of *operator precedence*. In the absence of parentheses, one associates arguments to operators according to the following rules.

Minus ($-$) has the highest precedence. Therefore $-a+b$ means $(-a)+b$. Next comes integer quotient ($\div$), remainder (**mod**) and multiplication ($*$), followed by the group consisting of addition ($+$) and subtraction ($-$), which is followed by the group $=$, $<$, $\leq$, $>$, $\geq$, and $\in$. Negation ($\neg$) comes next, followed by conjunction ($\wedge$), followed by disjunction ($\vee$). The quantifiers $\forall$ and $\exists$ are in the next group. Finally, the group consisting of implication ($\rightarrow$) and bi-implication ($\leftrightarrow$) has the lowest precedence.

As a result of these rules, $a * b + c$ means $(a{*}b){+}c$. The logical expression $p \wedge q \rightarrow r$ means $(p \wedge q) \rightarrow r$, and $\neg\ a \vee b \wedge c$ means $(\neg a) \vee (b \wedge c)$.

2.3 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 in a state-based specification such as this. A *relation* is a predicate on several arguments, the last of which is a state variable s. In the Mach specification, the relations are on 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; 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 $(y,\ s) \equiv \exists\, x\colon \mathrm{p}\,(x,\ y,\ s)$

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

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

3 Primitive 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 register 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*, n mapping from offsets to words, a unit of data. A task cannot 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 $\mathrm{taskp}\,(x,\ s)$ 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

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

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

Axiom 3.2

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

x satisfies *entityp* in state s if it is a member of one of the entity classes. The function *entities* gives the set of entities that exist in given state. The function *all-entities* contains the set of entities that *could* exist in a state. That is, it is the set of potential entities.

Definition 3.3

$$\begin{array}{ll} & \text{entityp}(x, s) \\ \equiv & \quad \text{taskp}(x, s) \\ & \vee\ \text{threadp}(x, s) \\ & \vee\ \text{portp}(x, s) \\ & \vee\ \text{messagep}(x, s) \\ & \vee\ \text{memoryp}(x, s) \\ & \vee\ \text{pagep}(x, s) \\ & \vee\ \text{procp}(x, s) \\ & \vee\ \text{procsetp}(x, s) \\ & \vee\ \text{devicep}(x, s) \end{array}$$

Axiom 3.4

$x \in \text{entities}(s) \leftrightarrow \text{entityp}(x, s)$

Axiom 3.5
entities $(s) \subseteq$ ALL-ENTITIES

We assume the existence of the constant NULL-PTR that identifies no entity.

Axiom 3.6
$\neg$ entityp (NULL-PTR, s)

One possible interpretation of the formula taskp $(x,\ s)$ is to think of s as an association list of names and entities. The predicate *taskp* holds on x when it is a name bound in s to an object of type *task*. One should think of the first argument to a recognizer as a reference to a unique entity, and not as an entity itself.

Implementation Note. The C implementation of Mach suggests an interpretation of the formula taskp $(x,\ s)$ which is similar to the association list interpretation. Think of s as the memory occupied by the kernel, and x as the address of a `task` structure as defined by the C code. taskp $(x,\ s)$ is implemented by a pointer to a task structure. 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`.

Webster [?] offers the following as a definition of the word *entity*.

> ... *the existence of a thing as contrasted with its attributes.*

This reflects our attitude about Mach entities. At the level of abstraction of this specification, a Mach entity has no contents. It is an element of an abstract data type and may engage in axiomatized relationships with other entities, and with elements of other sets, e.g., integers. The entity classes are only a subset of the elementary data types axiomatized in this paper. We introduce the others, for example *port names*, as needed in the presentation.

Implementation Note. A relation is typically implemented in the Mach code with pointers and flags. A field within a C structure which is used as a pointer to another implements a relation between instances of the structures. See Section 4 for a particularly simple example.

4 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 4.1
task-thread-rel $(t,\ \underline{th},\ s)$ WHERE
taskp $(t,\ s) \wedge$ threadp $(th,\ s)$

Definition 4.2
exists-owning-task $(th,\ s) \equiv \exists\, t$: task-thread-rel $(t,\ th,\ s)$

Axiom 4.3
exists-owning-task $(th,\ s) \rightarrow$ task-thread-rel (owning-task $(th,\ s),\ th,\ s)$

Axiom 4.4
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 4.5
taskp $(t,\ s) \wedge th \in$ threads $(t,\ s) \rightarrow$ threadp $(th,\ s)$

Implementation Note. *Taskp* is implemented by a pointer to a `task` structure, and *threadp* is implemented by a pointer to a `thread` structure. *Task-thread-rel* is implemented by the `task` field of a `thread` structure. This is a pointer from a thread to a task. 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.

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

5 Ports and Port Sets

5.1 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 5.1
$\mathcal{R} \equiv \{$`'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 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 5.2
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$)$

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

Definition 5.3
$\text{port-right-namep}\,(t,\ n,\ s) \equiv \exists\ p,\ R,\ i\colon \text{port-right-rel}\,(t,\ p,\ n,\ R,\ i,\ s)$

Axiom 5.4
$$\begin{aligned}&\ \ \text{port-right-namep}\,(t,\ n,\ s)\\ \rightarrow&\ \ \text{port-right-rel}\,(t,\ \text{named-port}\,(t,\ n,\ s),\ n,\ \text{port-rights}\,(t,\ n,\ s),\\ &\qquad\qquad\quad \text{port-right-refcount}\,(t,\ n,\ s),\ s)\end{aligned}$$

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 5.5
$(n = \text{NULLNAME}) \lor (n = \text{DEADNAME}) \rightarrow \neg\ \text{port-right-rel}\,(t,\ p,\ n,\ R,\ i,\ s)$

Axiom 5.6
$\neg\ \text{port-right-rel}\,(t,\ p,\ n,\ \emptyset,\ i,\ s)$

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

Axiom 5.7
$\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'receive}\},\ i,\ s) \rightarrow (i = \texttt{1})$

Axiom 5.8
$\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send-once}\},\ i,\ s) \rightarrow (i = \texttt{1})$

Axiom 5.9
$\text{port-right-rel}\,(t,\ p,\ n,\ R,\ i,\ s) \land (R = \{\texttt{'send},\ \texttt{'receive}\}) \rightarrow \texttt{2} \leq i$

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 5.10
$\text{s-right}\,(t,\ n,\ s) \equiv \text{port-right-namep}\,(t,\ n,\ s) \land \texttt{'send} \in \text{port-rights}\,(t,\ n,\ s)$

Definition 5.11
$$\begin{aligned}&\ \ \text{r-right}\,(t,\ n,\ s)\\ \equiv&\ \ \text{port-right-namep}\,(t,\ n,\ s) \land \texttt{'receive} \in \text{port-rights}\,(t,\ n,\ s)\end{aligned}$$

Definition 5.12

$$\begin{aligned}&\text{so-right}\,(t,\ n,\ s)\\ \equiv\ &\text{port-right-namep}\,(t,\ n,\ s) \land \texttt{'send-once} \in \text{port-rights}\,(t,\ n,\ s)\end{aligned}$$

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

Axiom 5.13

$$\begin{aligned}&\quad\ \ \text{s-right}\,(t,\ n_1,\ s)\\ &\quad \land\ \text{s-right}\,(t,\ n_2,\ s)\\ &\quad \land\ (\text{named-port}\,(t,\ n_1,\ s) = \text{named-port}\,(t,\ n_2,\ s))\\ &\to (n_1 = n_2)\end{aligned}$$

Axiom 5.14

$$\begin{aligned}&\quad\ \ \text{s-right}\,(t,\ n_1,\ s)\\ &\quad \land\ \text{r-right}\,(t,\ n_2,\ s)\\ &\quad \land\ (\text{named-port}\,(t,\ n_1,\ s) = \text{named-port}\,(t,\ n_2,\ s))\\ &\to (n_1 = n_2)\end{aligned}$$

Axiom 5.15

$$\begin{aligned}&\quad\ \ \text{r-right}\,(t,\ n_1,\ s)\\ &\quad \land\ \text{r-right}\,(t,\ n_2,\ s)\\ &\quad \land\ (\text{named-port}\,(t,\ n_1,\ s) = \text{named-port}\,(t,\ n_2,\ s))\\ &\to (n_1 = n_2)\end{aligned}$$

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 5.16

$$\text{so-right}\,(t,\ n,\ s) \to \neg\ \text{r-right}\,(t,\ n,\ s) \land \neg\ \text{s-right}\,(t,\ n,\ s)$$

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

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

[4] cf. [?], pg. 30

[5] cf. [?], pg. 25

Axiom 5.17

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

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 5.18

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

Axiom 5.19

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

Implementation Note. The data type `mach_port_t` implements $\mathcal{N}$. The constant `mach_port_null` is the implementation of NULLNAME, and `mach_port_dead` is the implementation of DEADNAME. The set of port rights granted to a task is maintained in an `ipc_space` structure—each task owns one and only one `ipc_space`. A computation on a `mach_port_t` and an `ipc_space` (done by the routine `ipc_entry_lookup`) identifies an `ipc_entry`, where a port capability is recorded. The `ie_bits` field of an `ipc_entry` encodes the right and reference count of a port right. The field `ie_object` contains a pointer to the port (an active `ipc_port` structure) involved in the right. If the port is inactive, the `ipc_entry` is interpreted as a dead right (see Section 5.3).

5.2 Port Sets

A *port set* is a named aggregation of port right names. The purpose of collecting these names is to permit a task to receive a message from any one of a number of ports. A port set is uniquely determined by its name in a task. The function *port-set* identifies the set of receive right names associated with task t and name n.

Relation 5.20
port-set-rel $(\underline{t},\ \underline{n},\ N,\ s)$ WHERE
taskp $(t,\ s) \wedge (n \in \mathcal{N}) \wedge (N \subseteq \mathcal{N})$

Definition 5.21
port-set-namep $(t,\ n,\ s) \equiv \exists\ N$: port-set-rel $(t,\ n,\ N,\ s)$

Axiom 5.22
port-set-namep $(t,\ n,\ s) \rightarrow$ port-set-rel $(t,\ n,$ port-set $(t,\ n,\ s),\ s)$

Neither NULLNAME nor DEADNAME may serve as the name of a port set.

Axiom 5.23
$(n = \text{NULLNAME}) \vee (n = \text{DEADNAME}) \rightarrow \neg$ port-set-rel $(t,\ n,\ N,\ s)$

The set of names involved in a port set relation must be a set of receive rights.[6]

Axiom 5.24
port-set-rel $(t,\ n,\ N,\ s) \wedge n_1 \in N \rightarrow$ r-right $(t,\ n_1,\ s)$

Port sets are mutually disjoint. In [**?**], page 31, this fact is stated by saying that a receive right can belong to only one port set.

Axiom 5.25
port-set-rel $(t,\ n_1,\ N_1,\ s) \wedge$ port-set-rel $(t,\ n_2,\ N_2,\ s) \wedge n_1 \neq n_2$
$\rightarrow ((N_1 \cap N_2) = \emptyset)$

If a receive-right belongs to a port set, the function *holding-port-set-name* names that port set.

Definition 5.26
in-port-set $(t,\ n_1,\ s)$
$\equiv \exists\ n$: port-set-namep $(t,\ n,\ s) \wedge n_1 \in$ port-set $(t,\ n,\ s)$

[6] cf. [**?**], pg. 31

Axiom 5.27

$$\begin{array}{ll} & \text{in-port-set}\,(t,\ n_1,\ s) \\ \rightarrow & \quad \text{port-set-namep}\,(t,\ \text{holding-port-set-name}\,(t,\ n_1,\ s),\ s) \\ & \wedge\ n_1 \in \text{port-set}\,(t,\ \text{holding-port-set-name}\,(t,\ n_1,\ s),\ s) \end{array}$$

Implementation Note. The Mach port set implementation is similar to the port right implementation. A `mach_port_t` identifies an `ipc_entry` contained in an `ipc_space`. The `ie_bits` field of an `ipc_entry` indicates whether or not the entry represents a port or port set. If so, the `ie_object` field points to an `ipc_pset` structure, which represents the port set.

5.3 Dead Rights

A task may hold a name which represents neither a port nor a port set. This is called a *dead right*. A dead right usually is created when a port involved in some port right is terminated. The name associated with that right becomes a *dead name*. As with a send right, a task can have multiple references to a dead right; the reference count of a dead right is a non-zero natural number less than the constant MAX-REFCOUNT. A dead right's reference count is unique. Therefore, we define *dead-right-refcount* to be a function on a task and a name.

Relation 5.28

$\text{dead-right-rel}\,(\underline{t},\ \underline{n},\ i,\ s)$ WHERE
$\text{taskp}\,(t,\ s) \wedge (n \in \mathcal{N}) \wedge (0 < i < \text{MAX-REFCOUNT})$

Definition 5.29

$\text{dead-right-namep}\,(t,\ n,\ s) \equiv \exists\ i\text{: dead-right-rel}\,(t,\ n,\ i,\ s)$

Axiom 5.30

$\text{dead-right-namep}\,(t,\ n,\ s) \rightarrow \text{dead-right-rel}\,(t,\ n,\ \text{dead-right-refcount}\,(t,\ n,\ s),\ s)$

Neither NULLNAME nor DEADNAME may serve as the name of a dead right.

Axiom 5.31

$(n = \text{NULLNAME}) \vee (n = \text{DEADNAME}) \rightarrow \neg\ \text{dead-right-rel}\,(t,\ n,\ i,\ s)$

Implementation Note. A dead right is implemented by an `ipc_entry` in which (a) the `ie_bits` field signals a dead name and the `ie_object` field equals `ip_null`, or (b) the `ie_bits` field signals a send or send-once right and the `ie_object` field points to an inactive `ipc_port` structure. Case (b) is lazily converted to (a) when discovered (in the routine `ipc_right_check`), so that an inactive port can eventually be reclaimed.

5.4 Local Names

For a given task, a name may be at most one of a port right name, a port set name or a dead name.

Axiom 5.32
port-set-namep $(t,\ n,\ s) \rightarrow \neg$ port-right-namep $(t,\ n,\ s)$

Axiom 5.33
dead-right-namep $(t,\ n,\ s)$
$\rightarrow \neg$ port-right-namep $(t,\ n,\ s) \wedge \neg$ port-set-namep $(t,\ n,\ s)$

A *local name* for a task is either a port right name, a port set name, or a dead name.

Definition 5.34
local-namep $(t,\ n,\ s)$
$\equiv$ port-right-namep $(t,\ n,\ s)$
$\vee$ port-set-namep $(t,\ n,\ s)$
$\vee$ dead-right-namep $(t,\ n,\ s)$

6 Virtual Memory

6.1 Introduction

Virtual memory is perhaps Mach's most subtle subsystem. Its implementation, using *copy-on-write* techniques to optimize the performance of copying and sharing memory, is complex. We address here not the implementation of virtual memory, but only the abstract view of memories, address spaces, and resident pages.

A Mach address space is a "sparsely populated indexed set of memory pages" ([**?**], page 36) associated with a task. To explain our formalization of Mach virtual memory, let us first take a look at the more familiar Unix view. In a Unix system, the entity which is "real" is a single virtual memory M large enough to hold all address spaces. M is, abstractly, a sequence of pages but it is usually implemented by a combination of RAM and backing store.

A virtual address va in task t is conceptually associated with a logical address la in M. Let's call this mapping vl, for *virtual-to-logical* translation.

$$vl : t, va \rightarrow la \vee \perp \tag{1}$$

$\perp$ represents failure. vl failure is caused only by the implementation error that there is not enough RAM and backing store to hold all of M. Abstractly, vl is a total function from virtual to logical addresses, i.e., there are no "address holes" in a task's address space.

In understanding Unix virtual address translation, we do not usually think about vl. Rather, we think about the mapping vp from virtual addresses to physical addresses that is part of the implementation of vl.

$$vp : t, va \rightarrow pa \vee \perp$$

vp maps a task's virtual address either to a physical address pa or to $\perp$, representing a page fault. vp together with page fault handlers implement the more abstract mapping vl.

Now let us look at Mach. Mach generalizes the Unix view to permit multiple memories M_1, M_2, etc. A virtual address in a task is conceptually associated with a logical address in one of the memories. The virtual-to-logical address translation therefore has the following signature.

$$vl : t, va \rightarrow m, la \vee \perp \tag{2}$$

where m is a memory and la is a logical address in that memory.

The virtual addresses in a Mach address space need not be associated with logical addresses in a single memory. Rather, different virtual addresses in a task can be associated with different memories. A virtual address may be associated with no memory and logical address, i.e., there can be holes in an address space. This is the *sparseness* referred to above. A page-aligned virtual address always maps to a page-aligned logical address in formula (1) above. This need not be the case in (2).

(2) suffices to characterize the most abstract view of Mach virtual memory. This view does not coincide, however, with the Mach kernel interface. Much of the responsibility for memory management lies outside the kernel, in so-called *external memory managers*. Each memory is associated with a memory manager that is responsible for maintaining the integrity of the memory's contents. When a page fault occurs, the kernel enters into a dialog with a memory manager to initialize the necessary resident pages. This dialog is part of the Mach 3.0 kernel interface. We have included in our model of Mach the idea of a resident page, and the notion that a resident page *represents* a segment of an abstract memory.

In the remainder of this section we present our model Mach virtual memory. Section 6.2 explains properties of abstract memories. We define relations that model a memory's contents, its control ports, and issues pertaining to its management — shadows and temporary bits. Section 6.3 explains how an abstract memory is mapped into a task's address space. Section 6.4 explains how a physical page may *represent* a region of an abstract memory, i.e., when the memory region is mapped in. Finally, in Section 6.5 we compose the aspects of the virtual memory model to specify a task's memory reference through its address map, through the mapped memory, to the appropriate representing page.

6.2 Abstract Memories

Memory Contents

Let $\mathcal{W}$ be a finite set of words. In this document a *word* is an undefined concept, but we assume that a representation of the number 0 occurs in $\mathcal{W}$. $\mathcal{W}$ is implemented by hardware bytes or words. The function *number-to-word* gives the representation of a non-negative number as a word, and *word-to-number* is its inverse.

The relation *memoryp* recognizes an abstract memory known to the kernel. (We consider a file that exists in a Mach system but is not open to some process not to be an abstract memory in the current state. It becomes an abstract memory when it is made known to the kernel.)

The content of an abstract memory is defined by the *store* relation. *Store* is a mapping from natural numbers to words. A natural number in the domain of a memory is called an *offset*.

It is either the case that a memory m and an offset o determine a word w, or else there is no word associated with m and o. We do not axiomatize a memory as a *sequence* of words, where every offset in some range is guaranteed to be associated with a word. If we did so, then *store-rel* would give the semantics of a Unix file. This constraint allows us to think of a memory as a partial function from natural numbers (offsets) to words. The function $m_S[o]$ denotes the word associated with offset o in memory m and state s. For $m_S[o]$ to be defined, o must be in the domain of the memory.

Relation 6.1
store-rel $(\underline{m},\ \underline{o},\ w,\ s)$ WHERE
memoryp $(m,\ s) \wedge (o \in \mathbf{N}) \wedge (w \in \mathcal{W})$

Definition 6.2
exists-mem-word $(m,\ o,\ s) \equiv \exists\ w$: store-rel $(m,\ o,\ w,\ s)$

Axiom 6.3
exists-mem-word $(m,\ o,\ s) \rightarrow$ store-rel $(m,\ o,\ m_S[o],\ s)$

In general, the contents of abstract memories are not implemented by the Mach kernel. They are implemented, rather, by user processes called *external memory managers*, or just *memory managers*. The kernel enters into a dialog with a memory manager to handle various aspects of memory management such as page faults, memory creation, and memory termination. Part of the purpose of a dialog is to maintain consistency between a memory and the kernel's cache.

We place no further constraints on a *store-rel* relation, since it is beyond the scope of this document to specify any processes above the level of the kernel. It is useful in specifying the kernel to have *store-rel* declared so that the dialog with an external memory manager can be described. In some cases, the kernel is responsible for initializing all or part of a memory's store. This occurs, for instance, when a range of zero-filled pages is allocated by the kernel call `vm_allocate`.

Control Ports

Managed memory is paged by some memory manager. It is associated with some special ports: an *object port*, a *control port* and a *name port*. These ports are important in the kernel/memory manager dialog. A memory manager holds a receive right on a memory's object port, and a send right on a memory's control port.[7] The kernel holds a send right on the object port and a receive right on the control and name ports.

A managed memory determines its object port, and vice versa. Similar constraints hold for control and name ports. We introduce *object-port*, *control-port* and *name-port* to be functions on a memory and a state that give a memory management port.

The functions *object-memory*, *control-memory*, and *name-memory* give the memory associated with a port.

Relation 6.4
object-port-rel $(\underline{m}, \overline{p}, s)$ WHERE
memoryp $(m, s) \land$ portp (p, s)

Relation 6.5
control-port-rel $(\underline{m}, \overline{p}, s)$ WHERE
memoryp $(m, s) \land$ portp (p, s)

Relation 6.6
name-port-rel $(\underline{m}, \overline{p}, s)$ WHERE
memoryp $(m, s) \land$ portp (p, s)

Definition 6.7
exists-object-port $(m, s) \equiv \exists\, p$: object-port-rel (m, p, s)

Axiom 6.8
exists-object-port $(m, s) \rightarrow$ object-port-rel $(m,$ object-port $(m, s), s)$

Definition 6.9
exists-object-memory $(p, s) \equiv \exists\, m$: object-port-rel (m, p, s)

[7] It may be the case that the actual EMM is on a different host. In this case the net message server is the receiver of the object port. The kernel does not treat this case specially – as far as the kernel is concerned, the net message server is the EMM for this memory entity.

Axiom 6.10
exists-object-memory $(p, s) \rightarrow$ object-port-rel (object-memory (p, s), p, s)

A memory is *managed* if and only if it has an object port.

Definition 6.11
managed $(m, s) \equiv$ exists-object-port (m, s)

No task other than the kernel task may hold a receive right on a control or name port.

Axiom 6.12
control-port-rel $(m, p, s) \wedge$ exists-receiver (p, s)
$\rightarrow$ (receiver $(p, s) = \textsc{kernel}$)

Axiom 6.13
name-port-rel $(m, p, s) \wedge$ exists-receiver $(p, s) \rightarrow$ (receiver $(p, s) = \textsc{kernel}$)

Implementation Note. *Memoryp* is implemented by a pointer to a `vm_object` structure. The `pager`, `pager_request` and `pager_name` fields of a `vm_object` name the object, control and name ports, respectively, of an abstract memory. The `ipc_kobject` field of one of these ports contains a back pointer to a `vm_object`. (The back pointer from a name port was added to the Mach 3.0 implementation in 1992.)

Temporary Memories

A memory may be *temporary.* This affects the behavior of the kernel, for example, when flushing dirty pages. A temporary memory may or may not be managed. A *persistent* memory is not temporary.

Relation 6.14
temporary-rel $(\underline{m}, s)$ WHERE
memoryp (m, s)

We've presented two orthogonal properties of abstract memories. A memory may or may not be *managed* and may or may not be *temporary.* A memory therefore falls into one of the following four categories.

1. **Unmanaged, Temporary**. A temporary memory is typically unmanaged. The kernel provides the initial represented values in physical memory. The kernel can cause an unmanaged temporary memory to become managed when there is a shortage of physical memory.[8]

2. **Managed, Temporary**. A managed temporary memory is usually managed by the default memory manager, but the identity of the default memory manager may change. Regardless of the identity of the manager of a managed, temporary memory, the kernel assumes that the pages representing such a memory need not be flushed back to the manager when the memory is deallocated.

3. **Managed, Persistent**. A persistent memory in always initially managed. The initial represented values for a persistent memory are provided through a dialog between the kernel and an external memory manager.

4. **Unmanaged, Persistent**. A persistent memory becomes unmanaged when its object port is terminated. Anomalous behavior of an external memory manager can cause this situation, for example, if the memory manager terminates or destroys its object port before the memory it manages is deallocated.

Implementation Note. A temporary memory can be created in the kernel, for example, by `vm_allocate`. A temporary memory is indicated by the logical *Or* of the `internal` and `temporary` bits of a `vm_object` structure. The temporary bit indicates that the object contents are of no interest after the object is unmapped, and can be discarded at that time instead of being returned to the manager. The internal bit indicates that the kernel created the object as opposed to it being mapped in from an external manager. `Internal` and `temporary` are currently duplicates (always have the same value).

Shadow Memories

When a memory m_1 *backs* a memory m_2, the kernel asks m_1's manager for a page if m_2's manager fails to supply it. This supports Mach's copy-on-write

[8] This is done in the routine vm_pageout_scan.

mechanism. Memory m_1 backs m_2 at an offset within m_1. We say that m_2 *shadows* m_1.

A memory may shadow at most one memory. Thus, *backing-rel* is a one-to-many relation on memories. The function *backing-memory* gives a memory's backing memory if one exists. The function *backing-offset* gives a memory's backing offset. WORDSIZE is the least upper bound on non-negative numbers that can be represented in a word.

Relation 6.15
backing-rel $(bm,\ \underline{sm},\ o,\ s)$ WHERE
memoryp $(bm,\ s) \wedge$ memoryp $(sm,\ s) \wedge (0 \leq o < \text{WORDSIZE})$

Definition 6.16
shadow-memoryp $(sm,\ s) \equiv \exists\ bm,\ o$: backing-rel $(bm,\ sm,\ o,\ s)$

Axiom 6.17
shadow-memoryp $(sm,\ s)$
$\rightarrow$ backing-rel (backing-memory $(sm,\ s)$, sm, backing-offset $(sm,\ s)$, s)

The predicate *backing-memoryp* is true if bm is a backing memory for a particular sm. The function *shadow-memories* gives the set of memories backed by a given memory.

Definition 6.18
backing-memoryp $(bm,\ sm,\ s) \equiv \exists\ o$: backing-rel $(bm,\ sm,\ o,\ s)$

Axiom 6.19
backing-memoryp $(bm,\ sm,\ s) \leftrightarrow sm \in$ shadow-memories $(bm,\ s)$

The *backing chain* of a memory m is the finite sequence of memories transitively related to m via *backing-rel*. When m shadows no memory, then the sequence is empty. Mach requires that there are no cycles in a backing chain. In other words, no memory is in its own backing chain.

Axiom 6.20
memoryp $(sm,\ s) \wedge (bc =$ backing-chain $(sm,\ s))$
$\rightarrow$ (shadow-memoryp $(sm,\ s) \rightarrow$ backing-memoryp $(bc_0,\ sm,\ s)$)
$\wedge$ ($(0 \leq i < (|bc| - 1))$
$\rightarrow$ (backing-chain $(bc_i,\ s) = bc_{(i\ +\ 1)..|bc|}$))

Axiom 6.21
$m \notin \text{backing-chain}\,(m,\, s)$

Implementation Note. The `shadow` field of a `vm_object` points to a memory's backing object. The `shadow_offset` field of a `vm_object` contains the offset into the backing object. The `shadowed` bit indicates that this memory is a backing object for some other memory. The Mach implementation guarantees that whenever a page must be copied from a backing memory m to its shadow memories, m has only one shadow memory. (See Section 4.4.5 of [**?**] for a discussion of virtual memory copying.) The function *shadow-memories* is implemented by the `copy` field of a `vm_object`, which points to the `vm_object` which is its only shadow-memory.

6.3 Address Spaces

The *map* relation associates a virtual page address (that is, a page-aligned virtual address less than ADDRESS-SPACE-LIMIT) *vpa* in task t with a number of attributes:

1. the contents of a memory m at offset o,
2. an inheritance value, and
3. a current and maximum protection.

The association with a memory and offset formalizes the virtual-to-logical mapping described in Section 6.1. The inheritance attribute is used to determine which regions of a parent task are mapped into a child task upon creation. The protection attributes enforce access control on regions of a virtual address space.

The task and virtual address of a map relation determine the other values in a *map-rel*. The function *mapped-memory* is the memory associated with task t at virtual address *vpa*. The function *mapped-offset* is the memory offset associated with task t at virtual address *vpa*.

The function *inheritance* gives the inheritance attribute of an instance of a map relation.

We define *protection* and *max-protection* to be the set of current and maximum protection attributes, respectively, for a virtual address in a task.

Relation 6.22
map-rel $(\underline{t},\ m,\ \underline{vpa},\ o,\ inh,\ CP,\ MP,\ s)$ WHERE
$\quad$ taskp $(t,\ s)$
$\wedge$ memoryp $(m,\ s)$
$\wedge$ $(0 \leq vpa < \text{ADDRESS-SPACE-LIMIT})$
$\wedge$ $(0 \leq o < \text{WORDSIZE})$
$\wedge$ $inh \in \mathcal{I}$
$\wedge$ $(CP \subseteq \mathcal{P})$
$\wedge$ $(MP \subseteq \mathcal{P})$

Axiom 6.23
vpa **mod** PAGESIZE $\neq 0 \rightarrow \neg$ map-rel $(t,\ m,\ vpa,\ o,\ inh,\ CP,\ MP,\ s)$

The set $\mathcal{I}$ defines the various inheritance options.

Definition 6.24
$\mathcal{I} \equiv \{$`'share`, `'copy`, `'none`$\}$

The set $\mathcal{P}$ defines the protection attributes.[9]

Definition 6.25
$\mathcal{P} \equiv \{$`'read`, `'write`, `'execute`$\}$

The current protection on a virtual address must not exceed its maximum protection.

Axiom 6.26
map-rel $(t,\ m,\ vpa,\ o,\ inh,\ CP,\ MP,\ s) \rightarrow (CP \subseteq MP)$

The task and virtual page address make a key.

Definition 6.27
allocated-vpa $(t,\ vpa,\ s) \equiv \exists\ m,\ o,\ inh,\ CP,\ MP$: map-rel $(t,\ m,\ vpa,\ o,\ inh,\ CP,\ MP,\ s)$

Axiom 6.28
$\quad$ allocated-vpa $(t,\ vpa,\ s)$
$\rightarrow$ map-rel $(t$, mapped-memory $(t,\ vpa,\ s)$, vpa, mapped-offset $(t,\ vpa,\ s)$,
inheritance $(t,\ vpa,\ s)$, protection $(t,\ vpa,\ s)$,
max-protection $(t,\ vpa,\ s)$, $s)$

[9] The same set is used to define locks on physical pages. See Section 6.4.

In the remainder of this section we introduce some useful derived functions. We say that a virtual address va is *allocated* within a task if the virtual address computed by truncating va to a page boundary occurs in a map relation. (The predicate *allocated-vpa*, above, is only true of a page-aligned virtual page address.)

Definition 6.29
trunc-page $(va) \equiv (va \div \text{PAGESIZE}) * \text{PAGESIZE}$

Definition 6.30
allocated $(t,\ va,\ s) \equiv$ allocated-vpa $(t,$ trunc-page $(va),\ s)$

A memory m is *mapped* if some offset o is mapped to an address within some task. (For a given task t, there may be several addresses where the memory is mapped.) *Mapping-tasks* returns the set of tasks which are currently mapping m.

Definition 6.31
task-maps-memoryp $(t,\ m,\ s)$
$\equiv\ \exists\ vpa$: allocated-vpa $(t,\ vpa,\ s) \land$ (mapped-memory $(t,\ vpa,\ s) = m$)

Axiom 6.32
memoryp $(m,\ s) \rightarrow (t \in$ mapping-tasks $(m,\ s) \leftrightarrow$ task-maps-memoryp $(t,\ m,\ s))$

Definition 6.33
mapped $(m,\ s) \equiv$ mapping-tasks $(m,\ s) \neq \emptyset$

Implementation Note. A task's address space is represented by a collection of data structures. The `map` field of a `task` structure is a pointer to a `vm_map`, which contains the topmost description of an address space. Conceptually, each page index is associated with a `vm_object` (the kernel's representation of an abstract memory). In fact, the information concerning consecutive indices that share the same `vm_object`, and protection and inheritance attributes (see subsequent sections of this paper), is coalesced into a single data structure called a `vm_map_entry`. The `object` field of `vm_map_entry` contains a pointer to a `vm_object`. The `offset` field is an offset into the abstract memory. The $\langle m, o\rangle$ pair of a *map* relation is determined by these two fields. The inheritance attribute of a virtual page index is contained in the `inheritance` field of a `vm_map_entry`. The protection attributes of a page index are contained in the `protection` and `m_protection` fields of a `vm_map_entry`.

6.4 Pages

A page is a finite sequence of words. The primitive relation that associates a page, offset and word is *page-word-rel*. All pages have the same size, a constant PAGESIZE, where $0 < \text{PAGESIZE}$. We use the notation pg-contents $(pg, s)_i$ to denote the ith word of page pg in state s, where $0 \leq i < \text{PAGESIZE}$.

Relation 6.34
page-word-rel $(\underline{pg},\ \underline{i},\ w,\ s)$ WHERE
pagep $(pg,\ s) \wedge (0 \leq i < \text{PAGESIZE}) \wedge (w \in \mathcal{W})$

Axiom 6.35

$$\begin{aligned}&\text{pagep}\,(pg,\ s) \wedge (0 \leq i < \text{PAGESIZE})\\ \rightarrow\ &(\text{pg-contents}\,(pg,\ s)_i = w \leftrightarrow \text{page-word-rel}\,(pg,\ i,\ w,\ s))\end{aligned}$$

A page may *represent* a portion of length PAGESIZE of a memory at a given offset. We say that a page *represents memory* if and only if it occurs in a *represents* relation with some memory and offset. A memory and offset are *represented* if they occur in a *represents* relation with some page. The page occuring in a *represents* relation uniquely determines a memory and an offset. The converse holds as well. A given offset within a memory may be represented by at most one page. Because of these uniqueness axioms, we can introduce the functions *represented-memory*, *represented-offset* and *representing-page*.

Relation 6.36
represents-rel $(\underline{pg},\ \overline{m},\ \overline{o},\ s)$ WHERE
pagep $(pg,\ s) \wedge$ memoryp $(m,\ s) \wedge (0 \leq o < \text{WORDSIZE})$

Definition 6.37
represented $(m,\ o,\ s) \equiv \exists\ pg$: represents-rel $(pg,\ m,\ o,\ s)$

Axiom 6.38
represented $(m,\ o,\ s) \rightarrow$ represents-rel (representing-page $(m,\ o,\ s),\ m,\ o,\ s)$

Definition 6.39
represents-memory $(pg,\ s) \equiv \exists\ m,\ o$: represents-rel $(pg,\ m,\ o,\ s)$

Axiom 6.40
represents-memory $(pg,\ s)$
$\rightarrow$ represents-rel $(pg,$ represented-memory $(pg,\ s)$, represented-offset $(pg,\ s),\ s)$

Implementation Note. We consider an entity that satisfies *pagep* to be a resident page. A *pagep* is implemented by a pointer to a `vm_page` structure. A `vm_page` contains the address of a physical page. The fields `object` and `offset` of a `vm_page` implement the $\langle m, o \rangle$ pair of a *represents* relation. The pages which represent segments of a given abstract memory are linked together. The header of this linked list is the `memq` field of a `vm_object`. This arrangement suggests two implementation invariants. First, the `object` field of a `vm_page` must point to the `vm_object` in which the `vm_page` is linked. Second, the `offset` field of a `vm_page` must be distinct from the offset of each other linked page. Failure to satisfy these invariants would imply violation of the constraints on the *represents* relation.

A page may be *dirty*, *precious* or *wired*. These attributes affect the dialog between the kernel and the external memory manager.

The *dirty* attribute characterizes the situation in which a value has been written to a page by a task, but the page is not yet flushed back to a memory manager. If space is short, pages that are not dirty can be deallocated at the whim of the kernel. The kernel assumes that the external memory manager has a copy of the unaltered page on its backing store. Dirty pages must be passed back to the external memory manager before they can be deallocated, so the external memory manager can update its store.

Relation 6.41
dirty-rel $(\underline{pg}, s)$ WHERE
pagep (pg, s)

In practice, the external memory manager might not retain a copy of information which is represented in a page. The external memory manager can mark a page as *precious*, which instructs the kernel that it must pass the page back to the external memory manager whether it is dirty or not.

Relation 6.42
precious-rel $(\underline{pg}, s)$ WHERE
pagep (pg, s)

A page which is *wired* cannot be evicted from memory under any circumstances. Only privileged tasks, for example device handlers and the default memory manager, can wire pages. A page can be wired multiple times up to an implementation-defined maximum MAX-WIRE-COUNT. A page's wire count is unique. Therefore, we define *wire-count* to be a partial function on a page.

Relation 6.43
wired-rel $(\underline{pg}, i, s)$ WHERE
pagep $(pg, s) \wedge (0 < i < \text{MAX-WIRE-COUNT})$

Definition 6.44
wired $(pg, s) \equiv \exists\, i$: wired-rel (pg, i, s)

Axiom 6.45
wired $(pg, s) \rightarrow$ wired-rel $(pg,$ wire-count $(pg, s), s)$

Implementation Note. A dirty page is identified by the `dirty` bit of a `vm_page` structure. The precious attribute is implemented by a `precious` bit, and the wired attribute is implemented by the `wired_count` field.

The Mach kernel provides a locking mechanism on pages. A lock associated with a page *prevents* access of the indicated type. Page locks are applied on request of an external memory manager. The set $\mathcal{P}$ defines the locking attributes which can be associated with a physical page.[10] The function *page-locks* gives the unique, non-empty set of locks applied to a page.

Relation 6.46
page-lock-rel$\,(\underline{pg},\ PR,\ s)$ WHERE
pagep$\,(pg,\ s) \wedge (PR \subseteq \mathcal{P})$

Definition 6.47
exists-page-locks$\,(pg,\ s) \equiv \exists\ PR$: page-lock-rel$\,(pg,\ PR,\ s)$

Axiom 6.48
exists-page-locks$\,(pg,\ s) \rightarrow$ page-lock-rel$\,(pg,$ page-locks$\,(pg,\ s),\ s)$

Definition 6.49
$\mathcal{P} \equiv \{$`'read`, `'write`, `'execute`$\}$

> **Implementation Note.** The `page_lock` field of a `vm_page` records a page's lock attribute.

In the remainder of this section we define the notions of a *zero page*, and of page equality.

A *zero segment* is a range of addresses within a page, each of which contains 0. The segment is identified by an offset o and length l.

Definition 6.50

$$
\begin{aligned}
& \text{zero-segment}\,(pg,\ o,\ l,\ s) \\
\equiv \quad & \text{pagep}\,(pg,\ s) \\
& \wedge\ (o \in \mathbf{N}) \\
& \wedge\ (l \in \mathbf{N}) \\
& \wedge\ o + l \leq \text{PAGESIZE} \\
& \wedge\ (\forall\, i\colon (\quad (o \leq i < (o + l)) \\
& \qquad\qquad \rightarrow (\text{pg-contents}\,(pg,\ s)_i = \text{number-to-word}\,(\mathtt{0}))))
\end{aligned}
$$

[10] The same set is used to define protection attributes on virtual addresses. See Section 6.3.

A *zero page* is one in which every location contains 0.

Definition 6.51
$\text{zero-page}\,(pg,\, s) \equiv \text{zero-segment}\,(pg,\, \texttt{0},\, \textsc{pagesize},\, s)$

Two pages are equal in the range of offsets $[o \ldots o{+}l)$ if *pg-read* is identical for both pages in that range. We call this notion *page-segment-equal*.

Definition 6.52

$$\begin{array}{ll} & \text{page-segment-equal}\,(pg_1,\, pg_2,\, o,\, l,\, s) \\ \equiv & \quad \text{pagep}\,(pg_1,\, s) \\ & \wedge\ \text{pagep}\,(pg_2,\, s) \\ & \wedge\ (o \in \mathbf{N}) \\ & \wedge\ (l \in \mathbf{N}) \\ & \wedge\ o + l \leq \textsc{pagesize} \\ & \wedge\ (\forall\, j\colon (\quad (o \leq j < (o + l)) \\ & \qquad\qquad \rightarrow (\text{pg-contents}\,(pg_1,\, s)_j = \text{pg-contents}\,(pg_2,\, s)_j))) \end{array}$$

Two pages are equal of they are *page-segment-equal* in the segments ranging from `0` to PAGESIZE.

Definition 6.53
$\text{page-equal}\,(pg_1,\, pg_2,\, s) \equiv \text{page-segment-equal}\,(pg_1,\, pg_2,\, \texttt{0},\, \textsc{pagesize},\, s)$

6.5 Task Memory Reference

In the abstract view of Mach's virtual memory system, a task does not have direct access to memory. Rather, it indirects through its address map, through the mapped memory, to an appropriate representing page. To summarize, we provide two partially specified state predicates m-wordp and va-wordp. The former is true of a memory entity, offset and word when there is a page and an index, related to the memory and offset, which contains the word at that index. The predicate *va-wordp* is true of a task, virtual address, and a word when the mapped memory of the task/address pair is in the *m-wordp* relation with the word.[11]

[11]If there is no such page, a page fault occurs.

Definition 6.54

$$
\begin{aligned}
& \text{m-wordp}\,(m,\ o_0,\ o,\ w,\ s) \\
\equiv\ & \quad \text{page-word-at-memory-offsetp}\,(m,\ o_0,\ o,\ w,\ s) \\
& \vee\ \text{page-word-in-shadow-chainp}\,(m,\ o_0,\ o,\ w,\ s)
\end{aligned}
$$

In the direct case, a page represents the memory at an offset o_0 that contains the offset o of interest. It is not a Mach requirement that the *represented-offset* of a page be at a page boundary, so it is possible for several pages to represent a given word. The offset o_0 identifes the particular page desired.

Definition 6.55

$$
\begin{aligned}
& \text{page-word-at-memory-offsetp}\,(m,\ o_0,\ o,\ w,\ s) \\
\equiv\ & \quad \text{represented}\,(m,\ o_0,\ s) \\
& \wedge\ \text{page-word-rel}\,(\text{representing-page}\,(m,\ o_0,\ s),\ o - o_0,\ w,\ s)
\end{aligned}
$$

If the memory/offset pair does not have a directly representing page, m may be a shadow memory. In this case we follow the memory's backing chain.

In a given state, if a shadow memory does not have the desired page in memory, but it is managed, then we cannot immediately follow the backing chain. The kernel must first query the external memory manager to determine if the page is swapped out at this point in the chain. We introduce the unspecified predicate *swapped-out* to indicate that, prior to this state, the kernel has queried the external memory manager. It is this quasi-temporal aspect that causes *m-wordp* to be partially specified.

Definition 6.56

$$
\begin{aligned}
& \text{page-word-in-shadow-chainp}\,(m,\ o_0,\ o,\ w,\ s) \\
\equiv\ & \quad \text{page-word-at-memory-offsetp}\,(m,\ o_0,\ o,\ w,\ s) \\
& \vee\ \quad \neg\ \text{represented}\,(m,\ o_0,\ s) \\
& \qquad \wedge\ \text{shadow-memoryp}\,(m,\ s) \\
& \qquad \wedge\ (\neg\ \text{managed}\,(m,\ s) \vee \neg\ \text{swapped-out}\,(m,\ o_0,\ s)) \\
& \qquad \wedge\ \text{page-word-in-shadow-chainp}\,(\text{backing-memory}\,(m,\ s), \\
& \qquad\qquad\qquad o_0 + \text{backing-offset}\,(m,\ s), \\
& \qquad\qquad\qquad o + \text{backing-offset}\,(m,\ s),\ w,\ s)
\end{aligned}
$$

Definition 6.57

$$
\begin{array}{ll}
 & \text{va-wordp}\,(t,\ va,\ w,\ s) \\
\equiv & \quad \text{allocated}\,(t,\ va,\ s) \\
 & \wedge\ \text{m-wordp}\,(\text{mapped-memory}\,(t,\ \text{trunc-page}\,(va),\ s), \\
 & \qquad\qquad \text{mapped-offset}\,(t,\ \text{trunc-page}\,(va),\ s), \\
 & \qquad\qquad\quad \text{mapped-offset}\,(t,\ \text{trunc-page}\,(va),\ s) \\
 & \qquad\qquad + \ (va - \text{trunc-page}\,(va)),\ w,\ s)
\end{array}
$$

Implementation Note. The mechanism described by *va-wordp* is optimized by the hardware page maps. These are managed by the *pmap* module, which is the interface between machine-dependent and machine independent operations. When a page fault occurs, the kernel follows data structures in a manner similar to those specified here.

7 Message Queues and Messages

7.1 Message Queues

A port contains a queue of messages. The relation *message-in-port* asserts that *mg* is the *ith* message in port p. The messages in a port are distinct, and a message occurs in at most one port. The function *containing-port* is a function on a message that gives its containing port if such a port exists, and *message-posn* gives the position of the message in its containing port. The sequence of messages associated with a port is given by the function *messages*, and the length of the sequence is given by the function *port-size*. We use the notation $\text{messages}\,(p,\ s)_i$ to denote the ith message in port p.

Relation 7.1

$\text{message-in-port-rel}\,(\underline{mg},\ \overline{p},\ \overline{i},\ s)$ WHERE
$\text{messagep}\,(mg,\ s) \wedge \text{portp}\,(p,\ s) \wedge (i \in \mathbf{N})$

Definition 7.2

$\text{exists-containing-port}\,(mg,\ s) \equiv \exists\ p,\ i\text{: message-in-port-rel}\,(mg,\ p,\ i,\ s)$

Axiom 7.3

$$
\begin{array}{ll}
 & \text{exists-containing-port}\,(mg,\ s) \\
\rightarrow & \text{message-in-port-rel}\,(mg,\ \text{containing-port}\,(mg,\ s),\ \text{message-posn}\,(mg,\ s),\ s)
\end{array}
$$

Axiom 7.4

$$\begin{array}{l} \quad \mathrm{portp}\,(p,\ s) \\ \rightarrow (\quad (0 \le i < \text{port-size}\,(p,\ s)) \wedge (\mathrm{messages}\,(p,\ s)_i = mg) \\ \quad\ \leftrightarrow \text{message-in-port-rel}\,(mg,\ p,\ i,\ s)) \end{array}$$

A port may be assigned a maximum message queue size. This is called the port's *qlimit*. The relation *message-qlimit-rel* associates a port with its qlimit. A port's qlimit is less than the constant MAX-QLIMIT.[12]

Relation 7.5

$\text{message-qlimit-rel}\,(\underline{p},\ i,\ s)$ WHERE
$\mathrm{portp}\,(p,\ s) \wedge (0 \le i < \text{MAX-QLIMIT})$

Definition 7.6

$\text{exists-qlimit}\,(p,\ s) \equiv \exists\, i\text{: message-qlimit-rel}\,(p,\ i,\ s)$

Axiom 7.7

$\text{exists-qlimit}\,(p,\ s) \rightarrow \text{message-qlimit-rel}\,(p,\ \mathrm{qlimit}\,(p,\ s),\ s)$

> **Implementation Note.** When a message is sent, the kernel creates a copy of the caller's data in an `ipc_kmsg` data structure.[a] We consider this internal representation of a message to be the implementation of *messagep*. The `ipc_mqueue` field of an `ipc_port` is a message queue. This field's data type is a structure that contains a header for a linked list of `ipc_kmsg`s.
>
> [a] cf. [?], pg. 32

7.2 Reply Ports

A message may be associated with a *reply port*. A reply port is the port to which the receiver of a message may send a reply. To do so, of course, the receiver must have a send right to the reply port. The relation indicates the transferred right. A message may be associated with at most one reply port. As a result, the functions *reply-port* and *reply-right* can be defined.

[12] Note that it is not a requirement that a port's size be less than its qlimit. Sending to a send-once right ignores the queue limit. Also, the queue limit may be decreased below its current value.

Relation 7.8
reply-port-rel $(\underline{mg}, p, r, s)$ WHERE
messagep $(mg, s) \wedge$ portp $(p, s) \wedge r \in \{\texttt{'send}, \texttt{'send-once}\}$

Definition 7.9
exists-reply-port $(mg, s) \equiv \exists\, p, r$: reply-port-rel (mg, p, r, s)

Axiom 7.10
exists-reply-port (mg, s)
$\rightarrow$ reply-port-rel $(mg$, reply-port (mg, s), reply-right $(mg, s), s)$

> **Implementation Note.** A pointer to the reply port of a message is carried in the `msgh_local_port` field of a message header.

7.3 Messages

A message is used to transmit port rights and data. A message is thought of as a finite sequence of data types induced by the relations discussed below. The index into the sequence (i in the relations below) is bounded by the constant MAX-MSG-SIZE.

A *transit right* is a relation on a message and a port. An assertion of a transit right means that message mg contains an access right r to port p (at index i of the message). A message and index determine a transit right.

Relation 7.11
transit-right-rel $(\underline{mg}, p, r, \underline{i}, s)$ WHERE
messagep $(mg, s) \wedge$ portp $(p, s) \wedge r \in \mathcal{R} \wedge (\mathtt{0} \leq i <$ MAX-MSG-SIZE$)$

Definition 7.12
exists-transit-right $(mg, i, s) \equiv \exists\, p, r$: transit-right-rel (mg, p, r, i, s)

Axiom 7.13
exists-transit-right (mg, i, s)
$\rightarrow$ transit-right-rel $(mg$, trans-port (mg, i, s), trans-right $(mg, i, s), i, s)$

No task may hold a receive right on a port whose receive right is being transmitted.

Axiom 7.14
transit-right-rel $(mg,\ p,\ \texttt{'receive},\ i,\ s) \rightarrow \neg$ exists-receiver $(p,\ s)$

Data can be transmitted in two ways: *in line* and *out of line*. *In line* transmission means that the data is placed in the destination task's message buffer and is copied in word by word. *Out of line* transmission means that pages containing the data are mapped somewhere into the destination task's address space.

Data is copied at the time a message is sent. In fact, data that is in transit conceptually is copied into a temporary abstract memory. An abstract memory that is in transit may not be mapped into any address space. (We state this constraint below.)

A *transit memory* is a relation on a message and a memory. An assertion of a transit memory means that message mg contains an abstract memory m to be inserted *in-line* or *out-of-line* in the receiving task. The offset and length parameters indicate a region of the memory which is "of interest" to the receiver. For out of line transmission the entire memory is mapped into the receiver's address space, but the receiver is informed that the data at offset o for length l is what was sent. The offset is smaller than PAGESIZE – it tells how much of the first page of data is to be ignored. The transit memory is the ith element of the message. A message and index determine a transit memory.

Relation 7.15
transit-memory-rel $(\underline{mg},\ m,\ a,\ o,\ l,\ \underline{i},\ s)$ WHERE
messagep $(mg,\ s)$
$\wedge$ memoryp $(m,\ s)$
$\wedge$ $a \in \{\texttt{'in-line},\ \texttt{'out-of-line}\}$
$\wedge$ $(0 \leq o < \text{PAGESIZE})$
$\wedge$ $(0 \leq l < \text{WORDSIZE})$
$\wedge$ $(0 \leq i < \text{MAX-MSG-SIZE})$

Definition 7.16
exists-transit-memory $(mg,\ i,\ s)$
$\equiv \exists\ m,\ a,\ o,\ l$: transit-memory-rel $(mg,\ m,\ a,\ o,\ l,\ i,\ s)$

Axiom 7.17

$$\begin{aligned}&\text{exists-transit-memory}\,(mg,\ i,\ s)\\ \rightarrow\ &\text{transit-memory-rel}\,(mg,\ \text{trans-memory}\,(mg,\ i,\ s),\ \text{trans-attribute}\,(mg,\ i,\ s),\\ &\qquad\qquad \text{trans-offset}\,(mg,\ i,\ s),\ \text{trans-length}\,(mg,\ i,\ s),\ i,\ s)\end{aligned}$$

No abstract memory occuring in a transit memory may be mapped into an address space. Furthermore, such a memory is temporary.

Axiom 7.18

$$\text{transit-memory-rel}\,(mg,\ m,\ a,\ o,\ l,\ i,\ s) \rightarrow \neg\ \text{mapped}\,(m,\ s) \wedge \text{temporary-rel}\,(m,\ s)$$

There are several flavors of null message elements. Because a message is used to transmit arguments to a remote procedure, there is a need at times to provide a null port right or null data as an argument. Additionally, a transit right may turn into a *dead transit right* if the port is killed while the message is queued. We define $\mathcal{M}$ to be the set of null message element types.

Definition 7.19

$$\mathcal{M} \equiv \{\texttt{'null-right},\ \texttt{'null-memory},\ \texttt{'dead-right}\}$$

The null message element relation identifies a component of a message that is null, and what flavor of null element the component is. The function *null-msg-tag* gives the tag associated with a null message element.

Relation 7.20

$$\begin{aligned}&\text{null-message-element-rel}\,(\underline{mg},\ tag,\ \underline{i},\ s)\ \ \text{WHERE}\\ &\text{messagep}\,(mg,\ s) \wedge tag \in \mathcal{M} \wedge (\texttt{0} \leq i < \text{MAX-MSG-SIZE})\end{aligned}$$

Definition 7.21

$$\begin{aligned}&\text{exists-null-msg-element}\,(mg,\ i,\ s)\\ \equiv\ &\exists\, tag\text{: null-message-element-rel}\,(mg,\ tag,\ i,\ s)\end{aligned}$$

Axiom 7.22

$$\begin{aligned}&\text{exists-null-msg-element}\,(mg,\ i,\ s)\\ \rightarrow\ &\text{null-message-element-rel}\,(mg,\ \text{null-msg-tag}\,(mg,\ i,\ s),\ i,\ s)\end{aligned}$$

A message is a finite sequence of transit rights, transit memories and null message elements. The function *message-contents* gives a representation of the sequence, and the function message-size $(mg,\ s)$ gives number of elements in a message.

Axiom 7.23

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{messagep}\,(mg,\, s) \wedge (0 \leq i < \text{message-size}\,(mg,\, s))} \\
\rightarrow & (& \text{message-contents}\,(mg,\, s)_i = \langle p,\, r\rangle \\
 & \leftrightarrow & \text{transit-right-rel}\,(mg,\, p,\, r,\, i,\, s)) \\
 & \wedge\ (& \text{message-contents}\,(mg,\, s)_i = \langle m,\, a,\, o,\, l\rangle \\
 & \leftrightarrow & \text{transit-memory-rel}\,(mg,\, m,\, a,\, o,\, l,\, i,\, s)) \\
 & \wedge\ (& \text{message-contents}\,(mg,\, s)_i = tag \\
 & \leftrightarrow & \text{null-message-element-rel}\,(mg,\, tag,\, i,\, s))
\end{array}
$$

Implementation Note. When a message is sent, the kernel creates a copy of the caller's data in an `ipc_kmsg` data structure.[a] We consider this internal representation of a message to be the implementation of *messagep*. Each element of a message is represented by a *type descriptor*, either the data structure `mach_msg_type_t` or the structure `mach_msg_type_long_t`. A type descriptor contains fields which determine the type of the element. The `msgt_name` field indicates whether an element is a port right or is data. If a port right, then `msgt_name` indicates which right is being transferred. If the element is data, then the `msgt_inline` bit indicates whether it is *in line* or *out of line* data.

[a] cf. [?], pg. 32

Implementation Note. A transit memory is implemented by a `vm_map_copy_t`, which is a variant record of one of three structures:

a list of `vm_entrys`. This data structure has the appearance of a segment of a task address space. Each entry has an associated `vm_object` data structure, which typically is a shadow memory backed by the memory object in the corresponding region of the sending task's address space.

a single vm_object. This option is an optimization used by the external memory manager pageout path.

a list of pages. This option is an optimization used by the external memory manager pagein path and by device drivers.

Note that our formalism most accurately describes the second option, which is not the typical case. If a sender sends a region of its address space mapped from several different memories, then (in the implementation) the receiver's address space has several memories mapped into it. This is visible through the `vm_region` kernel call. This possible inadequacy of our model deserves further thought.
The implementation also allows the transfer of an out-of-line array of port rights. Our formalization does not allow this.

We say that a port p_1 *transmits a receive right* for port p_2 if p_1 contains a message m that contains the transit right $\langle p_2, \texttt{'receive}\rangle$. The *transmission set* of a port p is the set of ports closed under transmission of a receive right.

Definition 7.24

$$\begin{aligned}&\text{transmits-r-right}\,(p_1,\, p_2,\, s)\\ \equiv\ &\exists\, mg,\, i,\, j:\quad \text{message-in-port-rel}\,(mg,\, p_1,\, i,\, s)\\ &\qquad\qquad \wedge\ \text{transit-right-rel}\,(mg,\, p_2,\, \texttt{'receive},\, j,\, s)\end{aligned}$$

Axiom 7.25

$$\begin{aligned}&p_1 \in \text{transmission-set}\,(p,\, s)\\ \leftrightarrow\ &\quad \text{transmits-r-right}\,(p,\, p_1,\, s)\\ &\vee\ (\exists\, p_2:\quad p_2 \in \text{transmission-set}\,(p,\, s)\\ &\qquad\qquad \wedge\ \text{transmits-r-right}\,(p_2,\, p_1))\end{aligned}$$

One can think of a port's transmission set as identifying a subgraph on the ports in a Mach state. Mach requires that there be no cycles in such a subgraph. In other words, a port is not in its own transmission set.

Axiom 7.26
$p \notin$ transmission-set $(p,\ s)$

> **Implementation Note.** Mach detects cycles in a transmission graph at the time a receive right is transmitted. All ports involved in the cycle are destroyed, since once a cycle is created no task can gain a right to any port in the cycle.

8 Physical Resources

8.1 Processor Sets and Processors

A processor set is a collection of processors. A processor set is a first class entity because of its relationships with other entities. For example, a processor set has a self port (see Section 9.4). Tasks and threads are also assigned to processor sets for scheduling purposes. The identity of a processor's processor set is a function of a processor and a state s. The function *proc-assigned-procset* is defined to be the unique processor set to which a processor is assigned, when such an assignment exists. The function *processors* is the set of processors associated with a processor set.

Relation 8.1
procset-proc-rel $(procset,\ \underline{proc},\ s)$ WHERE
procsetp $(procset,\ s) \wedge$ procp $(proc,\ s)$

Definition 8.2
exists-proc-assigned-procset $(proc,\ s)$
$\equiv \exists\ procset$: procset-proc-rel $(procset,\ proc,\ s)$

Axiom 8.3
exists-proc-assigned-procset $(proc,\ s)$
$\rightarrow$ procset-proc-rel (proc-assigned-procset $(proc,\ s)$, $proc,\ s)$

Axiom 8.4

procsetp $(procset,\ s)$
$\to (proc \in \text{processors}\,(procset,\ s) \leftrightarrow \text{procset-proc-rel}\,(procset,\ proc,\ s))$

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

Theorem 8.5

$\text{procsetp}\,(procset,\ s) \wedge proc \in \text{processors}\,(procset,\ s) \to \text{procp}\,(proc,\ s)$

> **Implementation Note.** A *procset* entity is implemented by a pointer to a `processor_set` structure, and *procp* is implemented by a pointer to a `processor` structure. We consider *procp* to be implemented by the kernel data structure, not the actual hardware. *procset-proc-rel* is implemented by the `processor_set` field of a `processor` structure. A `processor_set` contains the header of a linked list of processors assigned to this processor set. This suggests the implementation invariant that the `processor_set` field of a processor *proc* must point to the processor set in whose processor list *proc* is linked.

A processor set is assigned zero or more threads. The relation *procset-thread-rel* associates a thread with a processor set.

A thread belongs to at most one processor set. The identity of a thread's assigned processor set is a function of a thread and a state s. The function *thread-assigned-procset* is defined to be the unique processor set to which a thread is assigned, when such an assignment exists. The function *procset-threads* is the set of threads associated with a processor set.

Relation 8.6

$\text{procset-thread-rel}\,(procset,\ \underline{th},\ s)$ WHERE
$\text{procsetp}\,(procset,\ s) \wedge \text{threadp}\,(th,\ s)$

Definition 8.7

exists-thread-assigned-procset $(th,\ s)$
$\equiv \exists\ procset\text{: procset-thread-rel}\,(procset,\ th,\ s)$

Axiom 8.8

exists-thread-assigned-procset $(th,\ s)$
$\to \text{procset-thread-rel}\,(\text{thread-assigned-procset}\,(th,\ s),\ th,\ s)$

Axiom 8.9

$$\begin{array}{l} \quad \text{procsetp}\,(\mathit{procset},\ s) \\ \rightarrow (\mathit{th} \in \text{procset-threads}\,(\mathit{procset},\ s) \leftrightarrow \text{procset-thread-rel}\,(\mathit{procset},\ \mathit{th},\ s)) \end{array}$$

> **Implementation Note.** *procset-thread-rel* is implemented by the `processor_set` field of a `thread` structure. This is a pointer from a thread to a processor set. A `processor set` contains a header to a linked list of threads owned by the processor set. This suggests the implementation invariant that the `processor_set` field of a thread *th* must point to the processor set in whose thread list *th* is linked.

A processor set is also assigned zero or more tasks. This is used only for the default assignment for newly created threads in the task. The relation *procset-task-rel* and the derived functions *task-assigned-procset* and *processor-set-tasks* are developed in an analogous manner to the threads relations.

Relation 8.10

$$\begin{array}{l} \text{procset-task-rel}\,(\mathit{procset},\ \underline{t},\ s) \quad \text{WHERE} \\ \text{procsetp}\,(\mathit{procset},\ s) \wedge \text{taskp}\,(t,\ s) \end{array}$$

Definition 8.11

$$\begin{array}{l} \quad \text{exists-task-assigned-procset}\,(t,\ s) \\ \equiv \exists\ \mathit{procset}\colon \text{procset-task-rel}\,(\mathit{procset},\ t,\ s) \end{array}$$

Axiom 8.12

$$\begin{array}{l} \quad \text{exists-task-assigned-procset}\,(t,\ s) \\ \rightarrow \text{procset-task-rel}\,(\text{task-assigned-procset}\,(t,\ s),\ t,\ s) \end{array}$$

Axiom 8.13

$$\begin{array}{l} \quad \text{procsetp}\,(\mathit{procset},\ s) \\ \rightarrow (t \in \text{procset-tasks}\,(\mathit{procset},\ s) \leftrightarrow \text{procset-task-rel}\,(\mathit{procset},\ t,\ s)) \end{array}$$

> **Implementation Note.** The relation *procset-task-rel* is implemented by the `processor_set` field of a `task` structure. This is a pointer from a task to a processor set. A `processor set` contains a header to a linked list of tasks owned by the processor set. This suggests the implementation invariant that the `processor_set` field of a task *tk* must point to the processor set in whose task list *tk* is linked.

One processor set is known as the default processor set. It cannot be destroyed. The function *default-procset* is defined to return the default processor set, when one is set.

Relation 8.14
default-procset-rel ($procset$, s) WHERE
procsetp ($procset$, s)

Definition 8.15
exists-default-procset (s) $\equiv \exists$ $procset$: default-procset-rel ($procset$, s)

Axiom 8.16
exists-default-procset (s) $\rightarrow$ default-procset-rel (default-procset (s), s)

Implementation Note. The default processor set is initialized at bootstrap time in the global variable `default_pset` and is never changed.

One processor is known as the master processor. It is responsible for keeping time on a host. It may not be reassigned to a different processor set. The function *master-processor* is defined to return the master processor, when one is set.

Relation 8.17
master-processor-rel ($proc$, s) WHERE
procp ($proc$, s)

Definition 8.18
exists-master-processor (s) $\equiv \exists$ $proc$: master-processor-rel ($proc$, s)

Axiom 8.19
exists-master-processor (s) $\rightarrow$ master-processor-rel (master-processor (s), s)

Implementation Note. The master processor is initialized at bootstrap time in the global variable `master_processor` and is never changed.

8.2 Devices

The kernel interface to devices is very generic. A task gets access to a device port via a device master port. Thereafter, access to the port is via the device port in a device-specific protocol. We display the relation on devices and their special ports in Section 9.5.

Relation 8.20
master-device-port-rel $(p,\ s)$ WHERE
portp $(p,\ s)$

Definition 8.21
exists-master-device-port $(s) \equiv \exists\ p$: master-device-port-rel $(p,\ s)$

Axiom 8.22
exists-master-device-port (s)
$\rightarrow$ master-device-port-rel (master-device-port (s), s)

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

Axiom 8.23
master-device-port-rel $(p,\ s) \wedge$ exists-receiver $(p,\ s)$
$\rightarrow$ (receiver $(p,\ s) =$ KERNEL)

Implementation Note. A pointer to a `device` structure is the implementation of *devicep*. The master device port is initialized at bootstrap time in the global variable `master_device_port` and is never changed.

8.3 Hosts

We consider a state s to be the configuration of a given host. We may define functions on a state for obtaining information about a given host. For now, we are content merely to identify the function *host-time* that gives the global time kept by a host.

Axiom 8.24
host-time $(s) \in \mathbf{N}$

Implementation Note. A host data structure contains only pointers to the two host special ports (Section 9.6). The remainder of the data relevant to a host is distributed in global variables in the kernel task's address space.

9 Special Purpose Ports

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

We have already seen three special purpose ports: the memory management ports described in Section 6.2.

9.1 Task Special Ports

A task may be associated with a set of special ports.[13] A task self port (also called a task kernel port) identifies a task to the kernel and is used to request actions in behalf of a task. The task bootstrap port is typically used for locating services. An 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. The task exception port[14] is used by the kernel to convey information about exceptions.

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, and the function *self-task* to be the task with which a self port is associated.

The other kinds of task special ports need not be related to only one task. The functions *task-bport*, *task-sself*, and *task-eport* have axioms analogous to *task-self*.

[13]We are currently ignoring the registered ports.

[14]In later versions of Mach 3.0, the single exception port was replaced by a set.

Relation 9.1
task-self-rel $(\underline{t},\ \overline{p},\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

Relation 9.2
task-bport-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

Relation 9.3
task-sself-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

Relation 9.4
task-eport-rel $(\underline{t},\ p,\ s)$ WHERE
taskp $(t,\ s) \wedge$ portp $(p,\ s)$

Definition 9.5
exists-task-self $(t,\ s) \equiv \exists\ p$: task-self-rel $(t,\ p,\ s)$

Axiom 9.6
exists-task-self $(t,\ s) \rightarrow$ task-self-rel $(t,$ task-self $(t,\ s),\ s)$

Definition 9.7
exists-self-task $(p,\ s) \equiv \exists\ t$: task-self-rel $(t,\ p,\ s)$

Axiom 9.8
exists-self-task $(p,\ s) \rightarrow$ task-self-rel (self-task $(p,\ s),\ p,\ s)$

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

Axiom 9.9
task-self-rel $(t,\ p,\ s) \wedge$ exists-receiver $(p,\ s) \rightarrow$ (receiver $(p,\ s) =$ KERNEL)

Implementation Note. Pointers to a task's special ports exist in a task structure. The `ip_kobject` field of an `ipc_port` is a back pointer from a special port to the entity that it represents. The `ip_kotype` of a port gives a scalar value indicating what the type of special port. A port may play at most one special port role as defined by `ip_kotype`. Section 9.7 discusses this further.

9.2 Thread Special Ports

A thread has a self port, a sself port and an exception ports.[15]

The special ports of a thread are unique, and, additionally a self port is related to at most one thread. Therefore, we introduce the function *thread-self* to be the self port associated with a thread, and the function *self-thread* to be the thread with which a self port is associated. The functions *thread-eport* and *thread-sself* have axioms analogous to *thread-self*.

Relation 9.10
thread-self-rel $(\underline{th}, \overline{p}, s)$ WHERE
threadp $(th, s) \wedge$ portp (p, s)

Relation 9.11
thread-sself-rel $(\underline{th}, p, s)$ WHERE
threadp $(th, s) \wedge$ portp (p, s)

Relation 9.12
thread-eport-rel $(\underline{th}, p, s)$ WHERE
threadp $(th, s) \wedge$ portp (p, s)

Definition 9.13
exists-thread-self $(th, s) \equiv \exists\, p$: thread-self-rel (th, p, s)

Axiom 9.14
exists-thread-self $(th, s) \rightarrow$ thread-self-rel $(th,$ thread-self $(th, s), s)$

Definition 9.15
exists-self-thread $(p, s) \equiv \exists\, th$: thread-self-rel (th, p, s)

Axiom 9.16
exists-self-thread $(p, s) \rightarrow$ thread-self-rel (self-thread $(p, s), p, s)$

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

Axiom 9.17
thread-self-rel $(th1, p, s) \wedge$ exists-receiver (p, s)
$\rightarrow$ (receiver $(p, s) =$ KERNEL)

[15] In later versions of Mach 3.0, the single exception port was replaced by a set.

9.3 Processor Special Ports

A processor has one special port - its self port. The self port of a processor is unique. Conversely, a self port is related to at most one processor. We introduce the function *proc-self* to be the self port associated with a processor, and the function *self-proc* to be the processor with which a self port is associated.

Relation 9.18
proc-self-rel ($proc$, p, s) WHERE
procp ($proc$, s) $\wedge$ portp (p, s)

Definition 9.19
exists-proc-self ($proc$, s) $\equiv \exists\, p$: proc-self-rel ($proc$, p, s)

Axiom 9.20
exists-proc-self ($proc$, s) $\rightarrow$ proc-self-rel ($proc$, proc-self ($proc$, s), s)

Definition 9.21
exists-self-proc (p, s) $\equiv \exists\, proc$: proc-self-rel ($proc$, p, s)

Axiom 9.22
exists-self-proc (p, s) $\rightarrow$ proc-self-rel (self-proc (p, s), p, s)

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

Axiom 9.23
proc-self-rel ($th1$, p, s) $\wedge$ exists-receiver (p, s)
$\rightarrow$ (receiver (p, s) = KERNEL)

9.4 Processor Set Special Ports

A processor set has two special ports, a self port and a name port. A send right to the former gives a task the ability to change the processor set, but the latter may be used only to get information about the processor set.

The special ports of a processor set are unique. Conversely, a special port is related to at most one processor set. We introduce the function *procset-self* to be the self port associated with a processor-set, and the function *self-procset* to be the processor-set with which a self port is associated. The pair of functions *procset-name-port* and *name-port-procset* have analogous axioms.

Relation 9.24
procset-self-rel$\,(procset,\ p,\ s)$ WHERE
procsetp$\,(procset,\ s) \wedge$ portp$\,(p,\ s)$

Relation 9.25
procset-name-port-rel$\,(procset,\ p,\ s)$ WHERE
procsetp$\,(procset,\ s) \wedge$ portp$\,(p,\ s)$

Definition 9.26
exists-procset-self$\,(procset,\ s) \equiv \exists\ p$: procset-self-rel$\,(procset,\ p,\ s)$

Axiom 9.27
exists-procset-self$\,(procset,\ s)$
$\rightarrow$ procset-self-rel$\,(procset,$ procset-self$\,(procset,\ s),\ s)$

Definition 9.28
exists-self-procset$\,(p,\ s) \equiv \exists\ procset$: procset-self-rel$\,(procset,\ p,\ s)$

Axiom 9.29
exists-self-procset$\,(p,\ s) \rightarrow$ procset-self-rel (self-procset$\,(p,\ s),\ p,\ s)$

Only the kernel task may hold a receive right to a processor set's service ports.

Axiom 9.30
procset-self-rel$\,(procset,\ p,\ s) \wedge$ exists-receiver$\,(p,\ s)$
$\rightarrow$ (receiver$\,(p,\ s) = $ KERNEL)

Axiom 9.31
procset-name-port-rel$\,(procset,\ p,\ s) \wedge$ exists-receiver$\,(p,\ s)$
$\rightarrow$ (receiver$\,(p,\ s) = $ KERNEL)

9.5 Device Special Ports

A device has a special port called the *device port*. The device port of a device is unique. Conversely, a device port is related to at most one device. We introduce the function *device-port* to be the device port associated with a device, and the function *port-device* to be the device with which a device port is associated.

Relation 9.32
device-port-rel $(dev,\ p,\ s)$ WHERE
devicep $(dev,\ s) \wedge$ portp $(p,\ s)$

Definition 9.33
exists-device-port $(dev,\ s) \equiv \exists\ p$: device-port-rel $(dev,\ p,\ s)$

Axiom 9.34
exists-device-port $(dev,\ s) \rightarrow$ device-port-rel $(dev,$ device-port $(dev,\ s),\ s)$

Definition 9.35
exists-port-device $(p,\ s) \equiv \exists\ dev$: device-port-rel $(dev,\ p,\ s)$

Axiom 9.36
exists-port-device $(p,\ s) \rightarrow$ device-port-rel (port-device $(p,\ s),\ p,\ s)$

Only the kernel task may hold a receive right to a device's device port.

Axiom 9.37
device-port-rel $(th,\ p,\ s) \wedge$ exists-receiver $(p,\ s)$
$\rightarrow$ (receiver $(p,\ s) =$ KERNEL)

A task gets access to a device port by making a kernel request via the *master device* port (Section 8.2).

9.6 Host Special Ports

For hosts, the terms *host name port* and *host self port* are synonyms referring to the information port. The terms *host control port* or *privileged host port* refer to the port with which changes can be affected. We define the relations *host-name-port-rel* and *host-control-port-rel*, and the functions *host-name-port* and *host-control-port*. The relations are between a port and a state, and the functions have only a state argument.

Relation 9.38
host-control-port-rel $(p,\ s)$ WHERE
portp $(p,\ s)$

Definition 9.39
exists-host-control-port $(s) \equiv \exists\, p$: host-control-port-rel $(p,\ s)$

Axiom 9.40
exists-host-control-port $(s) \rightarrow$ host-control-port-rel (host-control-port (s), s)

Relation 9.41
host-name-port-rel $(p,\ s)$ WHERE
portp $(p,\ s)$

Definition 9.42
exists-host-name-port $(s) \equiv \exists\, p$: host-name-port-rel $(p,\ s)$

Axiom 9.43
exists-host-name-port $(s) \rightarrow$ host-name-port-rel (host-name-port (s), s)

Only the kernel task may hold a receive right to host service ports.

Axiom 9.44
host-control-port-rel $(p,\ s) \wedge$ exists-receiver $(p,\ s)$
$\rightarrow$ (receiver $(p,\ s)$ = KERNEL)

Axiom 9.45
host-name-port-rel $(p,\ s) \wedge$ exists-receiver $(p,\ s)$
$\rightarrow$ (receiver $(p,\ s)$ = KERNEL)

9.7 Uniqueness of Special Port Roles

We have identified a number of special roles to which a port can be assigned. There is a subset of these roles for which the kernel guarantees a port's assignment is unique - the kernel service ports plus memory control ports. For example, a port that is some task's self port may not also be the device master port. We state this axiom for the task self port as follows. Analogous axioms hold for the other relations.

Axiom 9.46

$$
\begin{array}{ll}
 & \text{task-self-rel}\,(t,\ p,\ s) \\
\rightarrow & \quad \neg\ \text{thread-self-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{proc-self-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{procset-self-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{procset-name-port-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{master-device-port-rel}\,(p,\ s) \\
 & \wedge\ \neg\ \text{device-port-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{host-control-port-rel}\,(p,\ s) \\
 & \wedge\ \neg\ \text{host-name-port-rel}\,(p,\ s) \\
 & \wedge\ \neg\ \text{object-port-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{control-port-rel}\,(x,\ p,\ s) \\
 & \wedge\ \neg\ \text{name-port-rel}\,(x,\ p,\ s)
\end{array}
$$

9.8 Notification Ports

A *notification port* is used for queuing notifications for certain classes of events. The kernel holds a send-once right on the notification port. Mach currently implements the following notification classes.

Port-Destroyed. A port-destroyed notification prevents a port from being destroyed. When an operation attempts to destroy the port, it is instead queued as a transit receive right in the notification message.

Dead-Name. A task is notified when a port is destroyed to which a task has a send right. The local name of the right is included in the notification.

No-Senders. A task is notified when a port to which it has a receive right has no send rights.

Message-Accepted. A task is notified when a message that it wishes to be sent to a port is finally delivered to the message queue.

Port-destroyed and message-accepted notifications will be deleted in subsequent versions of Mach. Therefore, we do not model them here.

Dead-Name Notifications

A dead-name notification is modeled by a relation on a port, a task, and a name. In *dn-notification-rel*, destruction of the port denoted by name n in task t causes a message containing n to be sent to port p. When the name of a task's capability on a port is changed (via `mach_port_rename`), the dead-name notification relation is modified to reflect the new name. At most one dead-name notification port is associated with a $\langle task, name \rangle$ pair. The function *dn-notification-port* gives this port.

Relation 9.47
dn-notification-rel $(p,\ t,\ n,\ s)$ WHERE
portp $(p,\ s) \wedge$ taskp $(t,\ s) \wedge (n \in \mathcal{N})$

Definition 9.48
exists-dn-notification-port $(t,\ n,\ s) \equiv \exists\ p$: dn-notification-rel $(p,\ t,\ n,\ s)$

Axiom 9.49
exists-dn-notification-port $(t,\ n,\ s)$
$\rightarrow$ dn-notification-rel (dn-notification-port $(t,\ n,\ s),\ t,\ n,\ s)$

Implementation Note. The `ip_dnrequests` field of an `ipc_port` is an array of $\langle port, name \rangle$ pairs. Each element of the array identifies a port to which a dead-name notification is to be sent when the port denoted by the containing `ipc_port` is destroyed. The notification contains the given name. An `ipc_entry` that represents a name for which there is a dead-name notification request contains an index into the `ip_dnrequests` field of the port denoted by the `ipc_entry`. When a task's name for a port is changed via the kernel call `mach_port_rename` the corresponding entry in `ip_dnrequests` is updated to contain the new name.

No-Senders Notifications

A no-senders notification is a relation on two ports. The first is the notification port. The second is the port for which notification of no senders is to be made. A port may have at most one no-senders notification port. The function *ns-notification-port* gives this port.

Relation 9.50
ns-notification-rel $(np,\ p,\ s)$ WHERE
portp $(np,\ s)$ $\wedge$ portp $(p,\ s)$

Definition 9.51
exists-ns-notification-port $(p,\ s) \equiv \exists\ np$: ns-notification-rel $(np,\ p,\ s)$

Axiom 9.52
exists-ns-notification-port $(p,\ s)$
$\rightarrow$ ns-notification-rel (ns-notification-port $(p,\ s)$, p, s)

> **Implementation Note.** The `ip_nsrequest` field of an `ipc_port` contains a pointer to a port's no-senders notification port.

10 Consistency of the Specification

This formalization of the Mach kernel contains over 400 axioms about some 250 functions. It is important to demonstrate that the axioms are not contradictory, i.e., that they are *consistent*. One does this by displaying known concepts that satisfy the axioms.

We have used Nqthm[16] to demonstrate the consistency of this specification. An Nqthm *constrain* event permits the introduction of a collection of new function symbols, and axioms which they must satisfy. The Nqthm user must also supply a "witness" to the axioms — that is, a collection of previously defined concepts that provide an interpretation of the new function symbols, and which satisfy the axioms. In this way, consistency of the axioms is proved, since it is demonstrated that there is *some* collection of concepts that satisfies the axioms.

We have supplied a very simple witness for the Mach state axioms. Interpret each of the relations as a predicate that returns the constant *false*. The only entity in the state is the kernel task. The axioms follow from this interpretation. This amounts to showing that an (almost) empty state satisfies the axioms. In other work, we hope to demonstrate that a more interesting witness — one that closely models the Mach implementation in C — satisfies the axioms, as well.

[16]The Boyer-Moore theorem prover. See the introduction of this report.

11 Conclusion

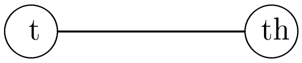
This report provides an axiomatic model of the primitive entities in a Mach kernel, and the relations on those entities which a Mach kernel must support. We have partially characterized nine primitive entities: task, thread, port, message, page, memory, processor, processor set, and device. This report represents an axiomatization in the Nqthm logic that contains over 400 axioms on approximately 250 functions. We have proved the consistency of these axioms by giving an interpretation of the new function symbols that satisfies the axioms. If this seems complicated one must remember that even so, it is an abstraction and simplification of the Mach kernel implementation in C.

There are several ways in which the reader may find fault with our mathematical model of Mach. First, we may have omitted some primitive entities which are deemed essential to Mach. Second, we may have omitted some relations on the included entities which are deemed essential. Finally, we may have made an error with respect to the intentions of the Mach designers in describing some relation. We hope that review of this paper will result in elimination of errors and a convergence of opinion on what must be described at the Mach interface.

Figures **??** and **??** give a visual representation of the entities and relations in a Mach kernel state. There is a node for each entity class. (Some nodes are duplicated in the two figures. This is done merely to minimize the number of intersecting lines. Three dimensions are required to do justice to the picture.) There is a link between nodes for each relation on two entities. Dangling labeled lines represent a relation involving only members of a single entity class. A Mach state can be thought of as a graph linking nodes (representing instances of the entity classes) via relations.

Alternatively, one may think of the axiomatization of a Mach kernel state in terms of a relational database. (See [**?**] for background on relational databases.) Each Mach relation introduced in in this report corresponds to a relation in the database. Each instance of a Mach relation (i.e., an application of a relation predicate to specific arguments) may be thought of as a tuple in the database.

In sum, we may think about the relations in several ways. For example, a task's ownership of a thread in a kernel state may be thought of as the assertion of some predicate, the existence of a tuple in a relational database, or as a link between two nodes in a state graph.

predicate	task-thread(t,th)
tuple	[t,th] in a task-thread relation
graph	

The predicate approach is useful for formal reasoning, and the graph approach for is useful for visualizing a state. As explored in [?], the database analogy is useful for addressing atomicity of transitions and concurrency.

We believe that this model of a legal state will make it possible to achieve more thorough analysis of Mach implementations and applications. A companion report [?] describes the identification of a collection of fine-grained atomic transitions in terms of which Mach kernel calls can be implemented. We intend to use this model as a basis for studying the correctness of the parallel implementation of Mach kernel calls. As a result of this work we expect the model to evolve somewhat as we fill in omissions, correct mistakes, and respond to design changes in Mach.

The ultimate source of information about Mach, as for many software systems, has been the source code. Description of the system and its underlying design principles exist in a number of published papers and CMU technical reports, e.g., [?], [?], [?].

The authors would like to thank the following for their input on this report: Todd Fine, Secure Computing Corporation (SCC), Sue Landauer, Trusted Information Systems (TIS), Spence Minear (SCC), Tim Redmond (TIS), Ed Schneider (SCC), and Matt Wilding (CLI).

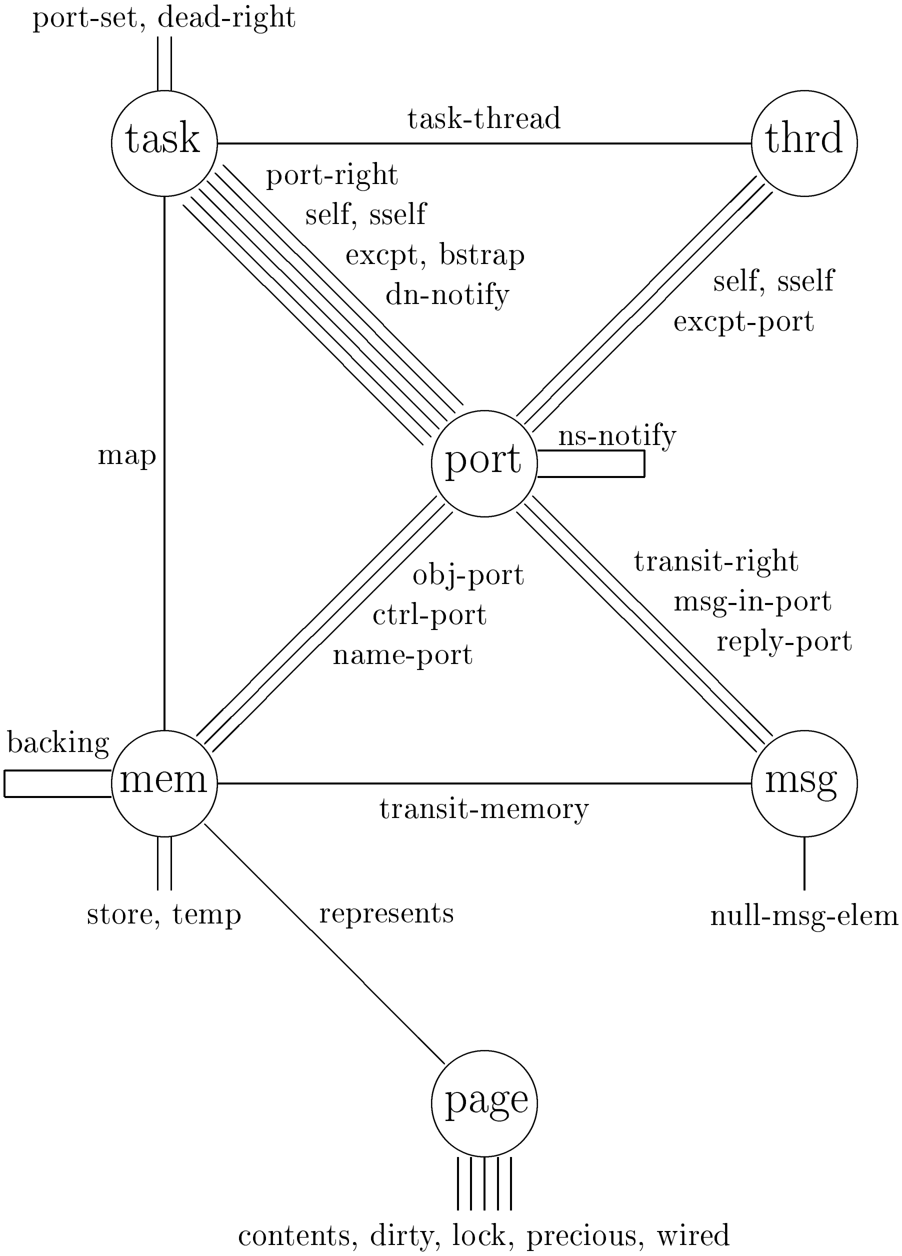


Figure 1: Mach Entities and Relations

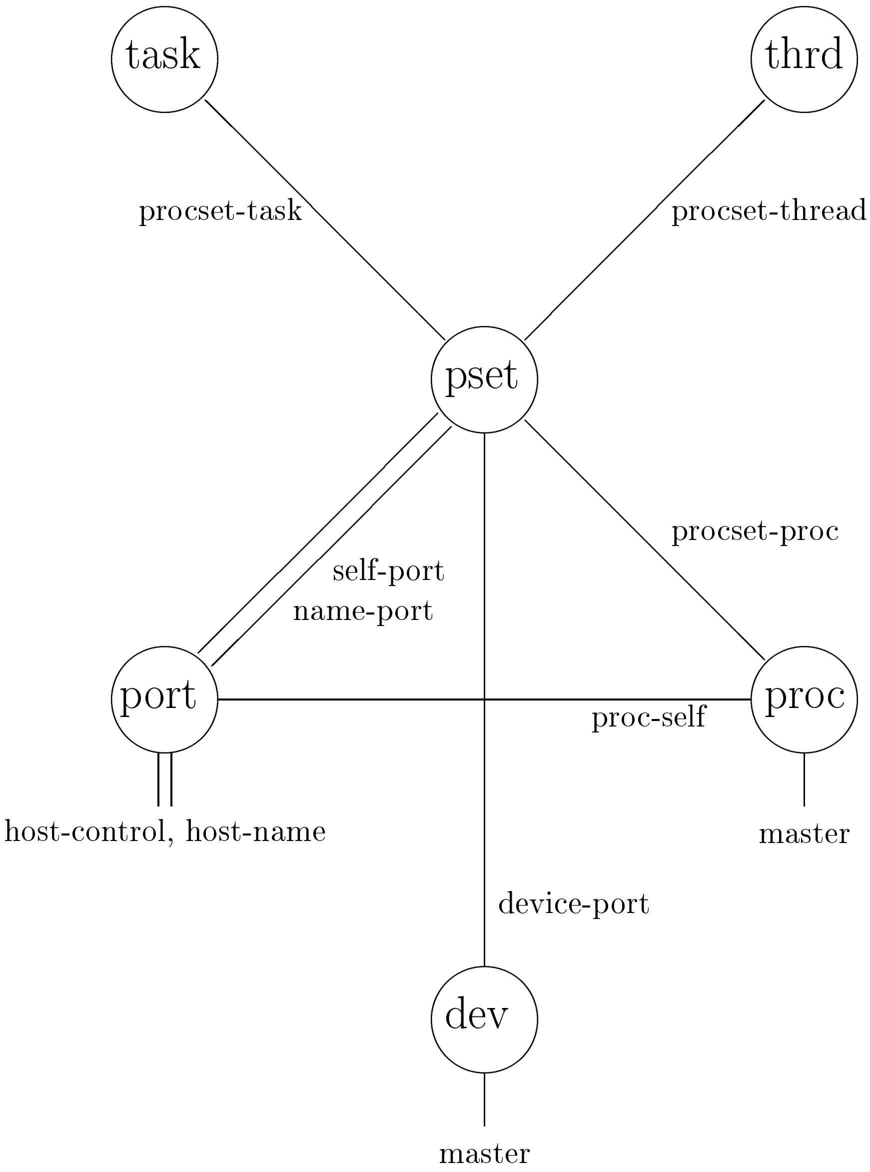


Figure 2: Mach Entities and Relations

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

[FR86] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[IEE90] *IEEE Standard Portable Operating System Interface for Computer Environments*. IEEE, New York, NY, 1990.

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

[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.

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

[Tev87] Avadis Tevanian, Jr. Architecture-independent virtual memory management for parallel and distributed environments: The Mach approach. Technical Report CMU-CS-88-106, Carnegie Mellon University, December 1987.

[Web87] *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster, Inc., 1987.

[You89] Michael Wayne Young. Exporting a user interface to memory management from a communication-oriented operating system. Technical Report CMU-CS-89-202, Carnegie Mellon University, November 1989.

Index