Part Number 83-0902024A001 Rev A
Version Date 4 December 1996

DTOS FORMAL TOP-LEVEL SPECIFICATION (FTLS)

CONTRACT NO. MDA904-93-C-4209
CDRL SEQUENCE NO. A005

Prepared for:
Maryland Procurement Office

Prepared by:

Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113

Authenticated by ________________________ Approved by ________________________
(Contracting Agency) (Contractor)

Date ________________________ Date ________________________

DTOS FORMAL TOP-LEVEL SPECIFICATION (FTLS)

Secure Computing Corporation

Abstract
This report formally describes a portion of the DTOS kernel.

Part Number	83-0902024A001 Rev A
Created	2 December 1994
Revised	4 December 1996
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/home/cmt/rev/dtos/docs/ftls/RCS/ftls.vdd,v 1.21 4 December 1996`

This document was produced using the TEX document formatting system and the LATEX style macros.

Contents

List of Figures

List of Tables

Section 1
Scope

1.1 Identification

This Formal Top Level Specification (FTLS) document presents a formal specification of a portion[1] of the prototype kernel developed on the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209.

1.2 System Overview

The DTOS prototype is an enhanced version of the CMU Mach 3.0 kernel that provides support for a wide variety of security policies by enforcing access decisions provided to it by a *security server*. By developing different security servers, a wide range of policies can be supported by the same DTOS kernel. By developing a security server that allows all accesses, the DTOS kernel behaves essentially the same as the CMU Mach 3.0 kernel. Although this is uninteresting from a security standpoint, it demonstrates the compatibility of DTOS with Mach 3.0.

By using appropriately developed security servers, the DTOS kernel can support interesting security policies such as MLS (multi-level security) and type enforcement. The first security server planned for development is one that enforces a combination of MLS and type enforcement.

1.3 Document Overview

The report is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Applicable Documents**, describes other documents that are relevant to this document.
- Section 3, **FTLS Overview**, provides an introduction to this document.
- Section 4, **Basic Kernel State Definition**, describes the data structures contained in the Mach kernel state.
- Section 5, **DTOS State Extensions**, describes extensions to the base Mach microkernel state that are needed to support the DTOS kernel.
- Section 6, **Kernel Execution Model**, describes the computational model used to model the DTOS kernel requests and the processing that is common to multiple DTOS requests.
- Section 7, **System Trap Requests**, describes a single request (**swtch**) that is invoked as a system trap.
- Section 8, **Port Requests**, describes a selection of the port manipulation requests.

[1] See Section 3 for a description the coverage.

- Section 9, **Thread Requests**, describes a selection of the thread manipulation requests.
- Section 10, **Virtual Memory Requests**, describes a selection of the virtual memory manipulation requests.
- Section 11, **Notes**, contains a partial list of acronyms and a small glossary for this document.
- Appendix A, **Bibliography**, provides the bibliographical information for the documents referenced in this document.
- Appendix B, **Z Extensions**, describes “extensions” to the Z specification language that are used in the DTOS FTLS.
- Appendix C, **IPC Requests**, describes the **mach_msg** request. This section has not yet been updated for DTOS. Currently, this section is a direct copy of the corresponding DTMach FTLS [5] section with minor changes required for DTOS sections that depend on this section and has been included only for easy reference.
- Appendix D, **Refinements of the State Model**, refines portions of the state model to a lower level of detail to model some of the data types and relationships that are used to implement the high-level abstract model described in the Basic Kernel State Definition and DTOS State Extensions chapters.

Section 2
Applicable Documents

The following document provides a high level description of the Mach microkernel:

- OSF Mach Kernel Principles [7]

The following documents provide a detailed description of the Mach and DTOS microkernels:

- OSF Mach 3 Kernel Interface [6]
- DTOS Kernel Interface Document (KID) [8]

The DTOS security policy model is described in

- DTOS Formal Security Policy Model (FSPM) [9]

Much of this document was derived from the following document:

- Formal Top Level Specification for Distributed Trusted Mach [5]

The following documents were used as additional sources of information on Mach:

- A Mathematical Model of the Mach Kernel: Entities and Relations (Draft) [2]
- A Mathematical Description of the Mach Kernel: Virtual Memory Services (Draft) [1]
- A Mathematical Model of the Mach Kernel: Port Services (Draft) [3]
- A Mathematical Model of the Mach Kernel: Task and Thread Services (Draft) [4]

Section 3
FTLS Overview

This document provides a *partial* formal top-level specification (FTLS) of the DTOS microkernel. We have made no attempt at complete coverage of the kernel interface. The FTLS includes

- A specification of the DTOS system state,
- The general properties of request execution common to most requests,
- One system trap request specification,
- 10 port request specifications,
- 21 thread request specifications, and
- Six virtual memory request specifications.

There are roughly 150 requests in DTOS, so this document covers approximately 25% of the DTOS kernel requests.

This document describes the system behavior both in English and in the Z formal specification language. Thus, readers who are unfamiliar with Z can simply ignore the formal Z specifications and read the English text.

Writing an FTLS is valuable because many behaviors of the system that have an impact on security can easily be overlooked in a less formal description. This is particularly true of behaviors which might be considered side effects of operations that have some other primary purpose. We have found this to be especially relevant for Mach since, by design, objects in the Mach microkernel have complex interactions. The process of formally specifying the behavior of a system frequently brings these behaviors to the surface.

3.1 Internal Consistency Within the FTLS

During the initial phases of the DTOS program, the primary goals of the FTLS were clarity, accuracy and completeness. As the program progressed, completeness in coverage was no longer a goal and the importance of accuracy was also diminished, leaving room for experimentation in the presentation of the FTLS. The result of this was that the FTLS is no longer internally consistent.[2]

There are two causes of inconsistency:

- Updates to the state model were not carried through to the request chapters. This only affects the virtual memory requests.
- The execution model was completely rewritten, and other than **mach_thread_self**, none of the request specifications were updated to conform with the new model. The specific effects of the new model on the request specifications is described in the following section.

[2] For further discussion of the evolution of the FTLS, the reasons for this evolution and general lessons learned while developing the FTLS, see the DTOS Lessons Learned Report[10].

3.2 Comments on Request Specifications

We have taken steps in writing this document to make the specifications easier to follow and to aid readers in locating desired information. First, requests are generally grouped based upon the type of kernel object to which the name given as the first parameter of the request is resolved (see the discussion of the client and kernel interfaces later in this section). Thus, requests whose first parameter is resolved to a thread (e.g., **thread_abort**), are in the Thread Requests chapter. Requests whose first parameter is resolved to a name space are in the Port Requests chapter, and requests whose first parameter is resolved to a memory map are in the VM Requests chapter.[3] There are, however, some exceptions to this rule. For example, the first parameter of a **thread_create** request is resolved to a task, but the request specification has been placed in the Thread Requests chapter since it is so intimately linked to the rest of the material in that chapter. This guideline does not apply at all to system trap requests many of which do not even have any parameters. So, these requests are in a separate chapter.

Second, we have attempted to make the structure of each chapter and each request specification as consistent as possible. Each chapter begins with an introduction that describes processing that is common or similar for multiple requests in the chapter.

The remainder of each chapter specifies the behavior of individual requests. Each of these specifications has the following structure:

1. Client Interface - the interface visible to the thread sending the request message. This includes a C Synopsis similar to that given in the Kernel Interface Document.
   (a) Input Parameters - the parameters included in the request message and the way in which the request in invoked.
   (b) Output Parameters - the parameters included in the reply message generated by the request (note that some parameters may occur in both lists).

   Editorial Note:
   The client interface is not consistent with the new execution model because the new execution model makes no attempt to model the client interface. The old model included hooks for such modeling though they were not integrated at all with the rest of the specification.

   ---

2. Kernel Interface - the interface visible to the kernel when it calls the kernel service routine.
   (a) Input Parameters - the input parameters included in the call.
   (b) Output Parameters - the output parameters included in the call (note that some parameters may occur in both lists).

   Editorial Note:
   The kernel interface is not consistent with the new execution model. The old execution model was based upon an abstraction which considered kernel requests to be received off of a message queue rather than "directly" from a trap as in the updated execution model.

   In the new execution model, the specification of the input interface is directly related to the extraction of parameters from a request ($ExtractRequest$) while specification of the output interface leads into the transition ($Return$) that describes the return from a kernel trap into user space.

   ---

[3] This draft of the FTLS does not explicitly represent name spaces and memory maps. It associates the names in a space and the regions in a map with the task that contains the name space and the memory map. However, the distinction between the task and the name space or memory map is still reflected in the division of the FTLS into chapters.

3. Request Criteria - the conditions that determine the behavior of the request (i.e., return values and state changes) in a given situation.

4. Return Values - a description of the value(s) returned in each possible situation as determined by the Request Criteria.

5. State Changes - a description of how the system state is changed by the request as determined by the Request Criteria.

6. Complete Request - ties together the multiple pieces of the specification including the common processing behavior specified in Section 6, **Kernel Execution Model**, and in the appropriate chapter introduction plus the return value and state change behavior described earlier in the given specification. This section is primarily of importance for the formal specification in Z. Readers who are ignoring the Z can skip the Complete Request sections entirely.

> *Editorial Note:*
> It is this section that is most directly influenced by the new execution model. With the new model, it should be possible to describe the total processing of a request more coherently in english as well as formally. See the specification of **mach_thread_self** for an example.

The following points should be made. First, some of the specifications contain a description of the parts of the system state that are invariant in the request. Other specifications do not contain such a description. In the cases where the invariants are specified, they have largely been inherited from the DTMach FTLS with only minor editing. They should not be considered to be highly reliable, particularly since the model of the system state presented in the DTOS FTLS is changed significantly from the model in the DTMach FTLS.

Second, the specification for each of the IPC-based kernel requests describes both the client and kernel interface. As stated above, the former describes the input and output parameters included in the invocation message and the reply message while the latter describes the input and output parameters included in the function call to the kernel routine. Typically, these parameters differ only in that some of the Mach names have been resolved to the object named (e.g., a thread, task, or port). We have not formally specified the relationship between these two interfaces.

3.3 Typographic Conventions

Finally, we have established typographic conventions for the identifiers used in the FTLS. In general, global objects in the specification contain capital letters while local objects are all in lower case. More specifically, the identifiers have the following forms:

- The names of Z schemas consist of capitalized words with no underscores between words (e.g., $SpecialThreadPorts$).

- Both schema components and variables consist of lower case words separated by underscores (e.g., $task\_self$).

- Global constants defined through axiomatic, generic and free type definitions have the first word capitalized, the remaining words in lower case and all the words separated by underscores (e.g., $Values\_partition$).

- All other identifiers (i.e., given types, free types, abbreviations and generic parameters) are printed in upper case with underscores to separate words (e.g., $PORT\_CLASS$).

A request name appearing in the text is set in bold face with words separated by underscores as they would be in a call within a program (e.g., **thread_create_secure**). In the formalization, each request has an identifier which is declared as an axiomatic global constant and is therefore typeset with the first word capitalized, the remaining words in lower case and all the words separated by underscores (e.g., $Thread\_create\_secure\_id$). One other typographic convention followed in this document is that components of the system state that are considered primitive (as opposed to being derived from some other piece or pieces of the system state) have their first character underlined (e.g., $\underline{t}ask\_self\_rel$).

Section 4
Basic Kernel State Definition

The following describes the data structures contained in the Mach kernel state. The organization of this section is as follows:

- Section 4.1, **Primitive Entities**, describes the primitive entities in Mach. Mach is an object-based system having these primitive entities as the defined objects.
- Section 4.2, **Process Management**, describes data structures associated with process management.
- Section 4.3, **Port Name Space**, describes data structures associated with task port name spaces.
- Section 4.4, **Ports**, describes data structures associated with ports.
- Section 4.5, **Notifications**, describes data structures associated with registered notifications.
- Section 4.6, **Special Ports**, describes the various classes of ports associated with the primitive entities.
- Section 4.7, **Total Send Rights**, describes the way in which send rights are counted in the kernel.
- Section 4.8, **Registered Rights**, describes the data structures used to record the set of port rights registered for a task.
- Section 4.9, **Memory System**, describes the data structures associated with the virtual memory system.
- Section 4.10, **Messages**, describes the data structures associated with messages.
- Section 4.11, **Processors and Processor Sets**, describes the data structures associated with processors and processor sets.
- Section 4.12, **Time**, describes the data structures associated with clocks.
- Section 4.13, **Devices**, describes the data structures associated with devices.

The model of Mach presented in this section consists of both primitive and derived notions. The derived notions provide no additional information about the Mach state beyond that embodied in the primitive notions. In the following sections, derived notions are noted as being conveniences. For example, Section 4.2.1 introduces the derived notion embodied by the function $threads$ to provide a more convenient representation for the primitive notion embodied by the relation $\underline{t}ask\_thread\_rel$. Although any statement about $threads$ can be reworded as a statement about $\underline{t}ask\_thread\_rel$, it is often more desirable to write the statement in terms of $threads$. In many cases, the choice of whether to view a structure as being primitive or derived is subjective. For example, others might prefer to view $\underline{t}ask\_thread\_rel$ as being derived from $threads$ instead of $threads$ being derived from $\underline{t}ask\_thread\_rel$.

As a convention, we underline the first letter in the identifier for each primitive structure in the Mach state. This is most useful when identifying which primitive structures are affected by DTOS services.

4.1 Primitive Entities

The primitive entities in Mach are:

Tasks — environments in which threads execute; a task consists of an address space, a port name space, and a set of threads

Threads — active entities comprised of an instruction pointer and a local register state

Ports — unidirectional communication channels between tasks

Messages — entities transmitted through ports

Memories — memory object representing a shared memory

Pages — logical units of memory; either a unit of physical memory or provided by a memory

Hosts — instances of the Mach kernel

Processors — devices capable of executing threads

Processor Sets — groups of processors, each belonging to a host, to which threads are assigned for scheduling

Devices — resources such as terminals and printers that can be used to transmit information between the system and its environment

Each of these primitive entities can be viewed as an abstract data type.

Mach Definition 1

$$[\, TASK, THREAD, PORT, MESSAGE, MEMORY, PAGE,$$
$$HOST, PROCESSOR, PROCESSOR\_SET, DEVICE \,]$$

At any given time, only certain primitive entities are present in the system. The sets $\underline{t}ask\_exists$, $\underline{t}hread\_exists$, $\underline{p}ort\_exists$, $\underline{m}essage\_exists$, $\underline{m}emory\_exists$, $\underline{p}age\_exists$, $\underline{p}roc\_exists$, $\underline{p}rocset\_exists$, and $\underline{d}evice\_exists$ denote the entities of each class that are present in the current system state.

Mach Definition 2

$$TaskExist \mathrel{\widehat=} [\underline{t}ask\_exists : \mathbb{P}\ TASK]$$
$$ThreadExist \mathrel{\widehat=} [\underline{t}hread\_exists : \mathbb{P}\ THREAD]$$
$$MessageExist \mathrel{\widehat=} [\underline{m}essage\_exists : \mathbb{P}\ MESSAGE]$$
$$MemoryExist \mathrel{\widehat=} [\underline{m}emory\_exists : \mathbb{P}\ MEMORY]$$
$$PageExist \mathrel{\widehat=} [\underline{p}age\_exists : \mathbb{P}\ PAGE]$$
$$ProcessorExist \mathrel{\widehat=} [\underline{p}roc\_exists : \mathbb{P}\ PROCESSOR]$$
$$ProcessorSetExist \mathrel{\widehat=} [\underline{p}rocset\_exists : \mathbb{P}\ PROCESSOR\_SET]$$
$$DeviceExist \mathrel{\widehat=} [\underline{d}evice\_exists : \mathbb{P}\ DEVICE]$$

Ip_null and Ip_dead are two special values in $PORT$ which are never in the set of existing ports. $port\_pointer$ consists of $\underline{p}ort\_exists$ plus the special values Ip_null and Ip_dead.

Mach Definition 3

$$Ip\_null, Ip\_dead : PORT$$
$$Ip\_null \neq Ip\_dead$$

Mach Definition 4

$$
\begin{array}{l}
\textit{PortExist} \\
\hline
\underline{p}ort\_exists : \mathbb{P}\, PORT \\
port\_pointer : \mathbb{P}\, PORT \\
\hline
Ip\_null \notin \underline{p}ort\_exists \\
Ip\_dead \notin \underline{p}ort\_exists \\
port\_pointer = \underline{p}ort\_exists \cup \{Ip\_null, Ip\_dead\}
\end{array}
$$

Mach Definition 5

$$
\begin{array}{l}
\textit{Exist} \\
\hline
TaskExist \\
ThreadExist \\
PortExist \\
MessageExist \\
MemoryExist \\
PageExist \\
ProcessorExist \\
ProcessorSetExist \\
DeviceExist
\end{array}
$$

Note that in the model, the kernel itself is viewed as an existing task and is denoted by $\underline{k}ernel$.

Mach Definition 6

$$
\begin{array}{l}
\textit{Kernel} \\
\hline
\underline{k}ernel : TASK \\
TaskExist \\
\hline
\underline{k}ernel \in \underline{t}ask\_exists
\end{array}
$$

4.2 Process Management

This section describes the data structures associated with process management. Multi-threaded processes are supported by allowing tasks to contain multiple threads.

4.2.1 Thread to Task Relationship

The relation $\underline{t}ask\_thread\_rel$ denotes the relationship between threads and tasks; a pair $(task, thread)$ is an element of $\underline{t}ask\_thread\_rel$ exactly when $thread$ is one of the threads contained in $task$. Each thread belongs to exactly one task. For convenience, the following additional notation is introduced:

- $owning\_task(thread)$ — the task to which $thread$ belongs
- $threads(task)$ — the set of threads belonging to $task$

Mach Definition 7

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad TasksAndThreads \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
TaskExist \\
ThreadExist \\
\underline{t}ask\_thread\_rel : TASK \leftrightarrow THREAD \\
owning\_task : THREAD \rightarrow\!\!\!\!\!+ TASK \\
threads : TASK \rightarrow\!\!\!\!\!+ \mathbb{P}\ THREAD \\
\hline
\mathrm{dom}\ \underline{t}ask\_thread\_rel \subseteq \underline{t}ask\_exists \\
\mathrm{ran}\ \underline{t}ask\_thread\_rel = \underline{t}hread\_exists \\
owning\_task = \underline{t}ask\_thread\_rel^{\sim} \\
threads \\
\quad = (\lambda\ task : TASK \\
\qquad \mid task \in \underline{t}ask\_exists \\
\qquad \bullet\ \underline{t}ask\_thread\_rel(\!|\{task\}|\!)) \\
\hline
\end{array}
$$

4.2.2 Execution Status

The execution status of a thread identifies whether a thread is running, waiting on an event, waiting uninterruptibly, and/or halted. A thread holds some subset of these characteristics at any point in time. The type RUN_STATES defines the possible thread characteristics. RUN_STATES has possible values $Running$, $Stopped$, $Waiting$, $Uninterruptible$ and $Halted$.

Mach Definition 8

$$RUN\_STATES ::= Running \mid Stopped \mid Waiting \mid Uninterruptible \mid Halted$$

The values of this type have the following meanings:

- $Running$ — The thread is either executing on a processor or is in a run queue waiting to execute.
- $Stopped$ — The thread has been asked to stop (and might have done so). A stopped thread does not execute any instructions.
- $Waiting$ — The thread is waiting for an event.
- $Uninterruptible$ — The thread is waiting uninterruptibly.
- $Halted$ — The thread is halted at what the kernel considers to be a “clean” point (i.e., it can be resumed properly).

The state $Uninterruptible$ does not imply the state $Waiting$. A $\underline{r}un\_state$ that includes the former but not the latter can result when the procedure `clear_wait` is called on a thread that is both $Uninterruptible$ and $Waiting$. The expression $\underline{r}un\_state(thread)$ indicates which of the above characteristics are held by an existing thread.

Each thread has an associated suspend count that determines whether the thread may execute user level instructions. This count is denoted by $\underline{t}hread\_suspend\_count(thread)$. A thread may execute such instructions only if the value of its suspend count is zero. It is a consequence of the operation of the system (and therefore is not stated as an axiom here) that only stopped threads have a suspend count greater than zero.

A thread may be swapped out. A thread that is swapped out has no kernel stack. The set of such threads is indicated by $\underline{s}wapped\_threads$. Some threads may be wired into the system. A wired thread may not be swapped out. The set $\underline{t}hreads\_wired$ denotes the set of wired threads. Certain threads are called idle threads. An idle thread is one that runs on a processor that has no user threads to run. (That is, the thread keeps the processor "idling".) User threads will not be marked as idle. We use $\underline{i}dle\_threads$ to denote the set of idle threads.

Mach Definition 9

$$
\begin{array}{l}
\textit{ThreadExecStatus} \\
\hline
\textit{ThreadExist} \\
\underline{r}un\_state : THREAD \rightarrow\!\!\!\!\!+ \mathbb{P}\, RUN\_STATES \\
\underline{t}hread\_suspend\_count : THREAD \rightarrow\!\!\!\!\!+ \mathbb{N} \\
\underline{s}wapped\_threads : \mathbb{P}\, THREAD \\
\underline{t}hreads\_wired : \mathbb{P}\, THREAD \\
\underline{i}dle\_threads : \mathbb{P}\, THREAD \\
\hline
\mathrm{dom}\, \underline{r}un\_state = \underline{t}hread\_exists \\
\mathrm{dom}\, \underline{t}hread\_suspend\_count = \underline{t}hread\_exists \\
\underline{s}wapped\_threads \subseteq \underline{t}hread\_exists \\
\underline{t}hreads\_wired \subseteq \underline{t}hread\_exists \\
\underline{i}dle\_threads \subseteq \underline{t}hread\_exists \\
\underline{t}hreads\_wired \cap \underline{s}wapped\_threads = \emptyset
\end{array}
$$

Each task also has a suspend count. The expression $\underline{t}ask\_suspend\_count(task)$ denotes the count associated with $task$. If this value is non-zero, then none of the threads in $task$ may execute regardless of their individual suspend counts.

Mach Definition 10

$$
\begin{array}{l}
\textit{TaskSuspendCount} \\
\hline
\underline{t}ask\_suspend\_count : TASK \rightarrow\!\!\!\!\!+ \mathbb{N}
\end{array}
$$

Review Note:
We should probably specify the relationships between $\underline{t}ask\_suspend\_count$, $\underline{t}hread\_suspend\_count$ and $\underline{r}un\_state$ here.

4.2.3 Priority Levels

Thread priority levels are used to determine thread execution scheduling priorities. Priority levels are represented as a subset of the integers (in particular by the numbers between 0 and 31 inclusive in current implementations). The set $Priority\_levels$ denotes the allowable priority levels. The relation $Lower\_priority$ indicates when a priority is lower than a second priority; in particular, (x, y) is an element of $Lower\_priority$ exactly when x is a lower priority than y. Since the implementation uses higher numbers to indicate lower priorities, x is lower than y when $x > y$. The relation $Higher\_priority$ is the inverse ordering indicating when a priority is higher than a second priority. The constants $Lowest\_possible\_priority$ and $Highest\_possible\_priority$ denote the maximum and minimum integers, respectively, in $Priority\_levels$.

Mach Definition 11

$$
\begin{array}{|l}
Priority\_levels : \mathbb{P}\,\mathbb{Z} \\
Lower\_priority, Higher\_priority : \mathbb{Z} \leftrightarrow \mathbb{Z} \\
Lowest\_possible\_priority, Highest\_possible\_priority : \mathbb{Z} \\
\hline
Lower\_priority \subset Priority\_levels \times Priority\_levels \\
\forall\, x, y : Priority\_levels \bullet (x, y) \in Lower\_priority \Leftrightarrow x > y \\
Higher\_priority = Lower\_priority^{\sim} \\
Lowest\_possible\_priority = max\ Priority\_levels \\
Highest\_possible\_priority = min\ Priority\_levels
\end{array}
$$

Using these relations, the minimum and maximum priorities in a set of priorities can be defined. These are denoted by $Lowest\_priority(priority\_set)$ and $Highest\_priority(priority\_set)$, respectively.

Mach Definition 12

$$
\begin{array}{|l}
Lowest\_priority, Highest\_priority : \mathbb{P}\,\mathbb{Z} \rightarrow\!\!\!\!\!+ \mathbb{Z} \\
\hline
\mathrm{dom}\ Lowest\_priority = \mathbb{P}_1\ Priority\_levels \\
\mathrm{ran}\ Lowest\_priority = Priority\_levels \\
\mathrm{dom}\ Highest\_priority = \mathbb{P}_1\ Priority\_levels \\
\mathrm{ran}\ Highest\_priority = Priority\_levels \\
\forall\, priority\_set : \mathbb{P}_1\,\mathbb{Z} \bullet Lowest\_priority(priority\_set) = max\ priority\_set \\
\forall\, priority\_set : \mathbb{P}_1\,\mathbb{Z} \bullet Highest\_priority(priority\_set) = min\ priority\_set
\end{array}
$$

There is a highest priority (equal to 12 in current implementations) normally granted to ordinary user threads. This priority is denoted by $Base\_user\_priority$.

Mach Definition 13

$$
\begin{array}{|l}
Base\_user\_priority : \mathbb{Z} \\
\hline
Base\_user\_priority \in Priority\_levels
\end{array}
$$

Three different types of priority values are associated with each thread.

- The expression $\underline{t}hread\_priority(thread)$ represents a base user-setable priority for $thread$.
- The expression $\underline{t}hread\_max\_priority(thread)$ represents the maximum value to which $\underline{t}hread\_priority(thread)$ can be set.
- The expression $\underline{t}hread\_sched\_priority(thread)$ represents the priority that the system uses to make scheduling decisions. This value is determined based upon $\underline{t}hread\_priority$ and the thread scheduling policy (discussed in Section 4.2.4), and is not directly set by the user. This value cannot exceed $\underline{t}hread\_priority(thread)$.

The priority level of a thread can temporarily be depressed by the request **swtch_pri** or **thread_switch** to allow other threads to run. When a thread is depressed, its priority is set to the lowest possible priority.[4] The set $\underline{d}epressed\_threads$ denotes those threads whose priority is currently depressed. The expression $\underline{p}riority\_before\_depression(thread)$ denotes the priority level $thread$ had before depression if $thread$'s priority level has been depressed and $\underline{t}hread\_priority(thread)$ otherwise.

Mach Definition 14

[4] Note, however, that not all threads having the lowest possible priority are depressed.

$$
\begin{array}{l}
\textbf{ThreadPri} \\
\hline
ThreadExist \\
\underline{t}hread\_priority : THREAD \rightarrowtail \mathbb{Z} \\
\underline{t}hread\_max\_priority : THREAD \rightarrowtail \mathbb{Z} \\
\underline{t}hread\_sched\_priority : THREAD \rightarrowtail \mathbb{Z} \\
\underline{d}epressed\_threads : \mathbb{P}\, THREAD \\
\underline{p}riority\_before\_depression : THREAD \rightarrowtail \mathbb{Z} \\
\hline
\operatorname{ran} \underline{t}hread\_priority \subseteq Priority\_levels \\
\operatorname{ran} \underline{t}hread\_max\_priority \subseteq Priority\_levels \\
\operatorname{ran} \underline{t}hread\_sched\_priority \subseteq Priority\_levels \\
\operatorname{ran} \underline{p}riority\_before\_depression \subseteq Priority\_levels \\
\underline{d}epressed\_threads \subseteq \underline{t}hread\_exists \\
\operatorname{dom} \underline{t}hread\_priority = \operatorname{dom} \underline{t}hread\_max\_priority = \operatorname{dom} \underline{t}hread\_sched\_priority \\
\quad = \operatorname{dom} \underline{p}riority\_before\_depression = \underline{t}hread\_exists \\
\forall\, thread : THREAD \mid thread \in \operatorname{dom} \underline{t}hread\_priority \\
\bullet\ (\underline{t}hread\_priority(thread), \underline{t}hread\_max\_priority(thread)) \notin Higher\_priority \\
\wedge\ (\underline{t}hread\_sched\_priority(thread), \underline{t}hread\_priority(thread)) \notin Higher\_priority \\
\forall\, thread : THREAD \mid thread \in \operatorname{dom} \underline{t}hread\_priority \setminus \underline{d}epressed\_threads \\
\bullet\ \underline{p}riority\_before\_depression(thread) = \underline{t}hread\_priority(thread) \\
\forall\, thread : THREAD \mid thread \in \underline{d}epressed\_threads \\
\bullet\ \underline{t}hread\_priority(thread) = Lowest\_possible\_priority
\end{array}
$$

Each existing task has an associated priority level, denoted by $\underline{t}ask\_priority(task)$, that is used to assign the initial priority for any thread created within the task.

Mach Definition 15

$$
\begin{array}{l}
\textbf{TaskPriority} \\
\hline
TaskExist \\
\underline{t}ask\_priority : TASK \rightarrowtail \mathbb{Z} \\
\hline
\operatorname{dom} \underline{t}ask\_priority = \underline{t}ask\_exists \\
\operatorname{ran} \underline{t}ask\_priority \subseteq Priority\_levels
\end{array}
$$

4.2.4 Scheduling Policies

Each thread has an associated scheduling policy, represented by $\underline{t}hread\_sched\_policy(thread)$. The type $SCHED\_POLICY$ represents the set of available scheduling policies. Examples of supported policies are Timesharing ($Timeshare$) and Fixed Priority ($Fixedpri$). Some scheduling policies have associated policy specific data that must be associated with each thread. For example, threads scheduled under the Fixed Priority policy must have an associated scheduling quantum. The type $SCHED\_POLICY\_DATA$ denotes policy specific scheduling data. The expression $\underline{t}hread\_sched\_policy\_data(thread)$ denotes any such policy specific data associated with $thread$. The set $\underline{s}upported\_sp$ indicates which scheduling policies are actually supported by a given Mach system. All Mach systems are required to support $Timeshare$ and each thread in a Mach system must be assigned one of the scheduling policies supported by the system.

Mach Definition 16

$$[SCHED\_POLICY, SCHED\_POLICY\_DATA]$$

$$\begin{array}{|l} Timeshare, Fixedpri : SCHED\_POLICY \\ \hline Timeshare \neq Fixedpri \end{array}$$

Mach Definition 17

$$\begin{array}{|l} \textit{ThreadSchedPolicy} \\ \hline ThreadExist \\ \underline{t}hread\_sched\_policy : THREAD \leftrightarrow SCHED\_POLICY \\ \underline{t}hread\_sched\_policy\_data : THREAD \leftrightarrow SCHED\_POLICY\_DATA \\ \underline{s}upported\_sp : \mathbb{P}\, SCHED\_POLICY \\ \hline \mathrm{dom}\, \underline{t}hread\_sched\_policy\_data \subseteq \mathrm{dom}\, \underline{t}hread\_sched\_policy = \underline{t}hread\_exists \\ Timeshare \in \underline{s}upported\_sp \\ \mathrm{ran}\, \underline{t}hread\_sched\_policy \subseteq \underline{s}upported\_sp \\ \hline \end{array}$$

4.2.5 Instruction Pointer

The set $VIRTUAL\_ADDRESS$ is used to denote the set of virtual addresses. These addresses are assumed to be ordered in some manner with Vm_start and Vm_end denoting, respectively, the smallest and largest addresses.

Mach Definition 18

$$[VIRTUAL\_ADDRESS]$$

$$\begin{array}{|l} Vm\_start, Vm\_end : VIRTUAL\_ADDRESS \end{array}$$

Each thread has an associated instruction pointer indicating the address at which the thread is currently executing. The expression $\underline{i}nstruction\_pointer(thread)$ denotes $thread$'s current instruction pointer.

Mach Definition 19

$$\begin{array}{|l} \textit{ThreadInstruction} \\ \hline \underline{i}nstruction\_pointer : THREAD \leftrightarrow VIRTUAL\_ADDRESS \\ \hline \end{array}$$

4.2.6 Emulation Environment

Mach supports binary compatibility by allowing tasks to establish user-level handlers for system calls. This is accomplished by associating an *emulation vector* with each task. Each entry in an emulation vector specifies a system call and a virtual address. Whenever the task executes a system call that has an entry in the emulation vector, the code at the specified virtual address for the system call is executed rather than the system call. The expression $\underline{e}mulation\_vector(task)$ denotes $task$'s emulation vector.

Mach Definition 20

$$\begin{array}{|l} \textit{EmulationVector} \\ \hline TaskExist \\ \underline{e}mulation\_vector : TASK \leftrightarrow \mathbb{N} \leftrightarrow VIRTUAL\_ADDRESS \\ \hline \mathrm{dom}\, \underline{e}mulation\_vector = \underline{t}ask\_exists \\ \hline \end{array}$$

4.2.7 Sampling

Any thread or task may be sampled. This causes the instruction pointer to be recorded in a buffer during clock interrupts or page faults if the thread or task is currently executing. The type $SAMPLE$ represents the sampling information that is collected, and type $SAMPLE\_TYPES$ represents information that determines at which times during execution samples are collected for a given thread or task.

There are six recognized sample types. They are:

- $Sample\_periodic$ — each clock interrupt
- $Sample\_vm\_zfill\_faults$ — zero-filling a virtual memory page
- $Sample\_vm\_reactivation\_faults$ — reactivating a virtual memory page
- $Sample\_vm\_pagein\_faults$ — bringing a virtual memory page in
- $Sample\_vm\_cow\_faults$ — virtual memory copy-on-write faults
- $Sample\_vm\_faults\_any$ — all virtual memory page faults. This includes miscellaneous faults beyond the above mentioned four types of virtual memory faults.

These values comprise the elements of the set $Recognized\_sample\_types$.

Mach Definition 21

$$[SAMPLE, SAMPLE\_TYPES]$$

$$\begin{array}{|l}
Sample_periodic, Sample_vm_zfill_faults, \\
Sample_vm_reactivation_faults, Sample_vm_pagein_faults, \\
Sample_vm_cow_faults, Sample_vm_faults_any : SAMPLE_TYPES \\
Recognized_sample_types : \mathbb{P}\ SAMPLE_TYPES \\
\hline
\langle Sample_periodic, Sample_vm_zfill_faults, \\
Sample_vm_reactivation_faults, Sample_vm_pagein_faults, \\
Sample_vm_cow_faults, Sample_vm_faults_any\rangle \\
\quad Values_partition\ Recognized_sample_types
\end{array}$$

For convenience, $SAMPLE\_VM\_FAULTS$ is used as the combination of the sample types $Sample\_vm\_zfill\_faults$, $Sample\_vm\_reactivation\_faults$, $Sample\_vm\_pagein\_faults$ and $Sample\_vm\_cow\_faults$.

There is a maximum number of samples (determined by the buffer size) that can be kept for any thread or task. This maximum is represented by $Max\_samples$.

Mach Definition 22

$$SAMPLE\_VM\_FAULTS == \{Sample\_vm\_zfill\_faults, Sample\_vm\_reactivation\_faults, \\ Sample\_vm\_pagein\_faults, Sample\_vm\_cow\_faults\}$$

$$\begin{array}{|l} Max\_samples : \mathbb{N}_1 \end{array}$$

The set $\underline{s}ampled\_threads$ denotes the set of threads that are currently being sampled. For each sampled thread there is a set of sample types, denoted by $\underline{t}hread\_sample\_types(thread)$, indicating when a sample should be taken for the thread. Each sample taken for a thread is assigned a unique sequence number. The expression $\underline{t}hread\_sample\_sequence\_number(thread)$ denotes the sequence number of the most recent sample for a thread (or zero if no samples have been collected). The expression $\underline{t}hread\_samples(thread)$ denotes the currently stored samples for $thread$. Each sample is stored with an associated sample number. Only the $Max\_samples$ most recent samples are retained.

Mach Definition 23

$$
\begin{array}{l}
\textit{ThreadSampling} \\
\hline
ThreadExist \\
\underline{s}ampled\_threads : \mathbb{P}\ THREAD \\
\underline{t}hread\_sample\_types : THREAD \pfun \mathbb{P}\ SAMPLE\_TYPES \\
\underline{t}hread\_sample\_sequence\_number : THREAD \pfun \mathbb{N} \\
\underline{t}hread\_samples : THREAD \pfun (\mathbb{N} \pfun SAMPLE) \\
\hline
\underline{s}ampled\_threads \subset \underline{t}hread\_exists \\
\mathrm{dom}\ \underline{t}hread\_sample\_types = \underline{s}ampled\_threads \\
\mathrm{dom}\ \underline{t}hread\_sample\_sequence\_number = \underline{s}ampled\_threads \\
\mathrm{dom}\ \underline{t}hread\_samples = \underline{s}ampled\_threads \\
\forall\ smpls : \mathbb{N} \pfun SAMPLE;\ thread : THREAD; \\
\quad num, high : \mathbb{N} \\
\mid (thread, smpls) \in \underline{t}hread\_samples \\
\quad \wedge\ high = \underline{t}hread\_sample\_sequence\_number(thread) \\
\quad \wedge\ num = min\ \{high, Max\_samples\} \\
\bullet\ \mathrm{dom}\ smpls = high - num + 1 \mathinner{..} high
\end{array}
$$

The same sampling information is kept for tasks.

Mach Definition 24

$$
\begin{array}{l}
\textit{TaskSampling} \\
\hline
TaskExist \\
\underline{s}ampled\_tasks : \mathbb{P}\ TASK \\
\underline{t}ask\_sample\_types : TASK \pfun \mathbb{P}\ SAMPLE\_TYPES \\
\underline{t}ask\_sample\_sequence\_number : TASK \pfun \mathbb{N} \\
\underline{t}ask\_samples : TASK \pfun (\mathbb{N} \pfun SAMPLE) \\
\hline
\underline{s}ampled\_tasks \subset \underline{t}ask\_exists \\
\mathrm{dom}\ \underline{t}ask\_sample\_types = \underline{s}ampled\_tasks \\
\mathrm{dom}\ \underline{t}ask\_sample\_sequence\_number = \underline{s}ampled\_tasks \\
\mathrm{dom}\ \underline{t}ask\_samples = \underline{s}ampled\_tasks \\
\forall\ smpls : \mathbb{N} \pfun SAMPLE;\ task : TASK; \\
\quad num, high : \mathbb{N} \\
\mid (task, smpls) \in \underline{t}ask\_samples \\
\quad \wedge\ high = \underline{t}ask\_sample\_sequence\_number(task) \\
\quad \wedge\ num = min\ \{high, Max\_samples\} \\
\bullet\ \mathrm{dom}\ smpls = high - num + 1 \mathinner{..} high
\end{array}
$$

4.2.8 Thread Time Statistics

The system records time statistics for each thread. The following information is recorded:

- $\underline{u}ser\_time(thread)$ — the total user run time for $thread$
- $\underline{s}ystem\_time(thread)$ — the total system run time for $thread$

- $\underline{c}pu\_time(thread)$ — $thread$'s scaled CPU usage
- $\underline{s}leep\_time(thread)$ — the amount of time for which $thread$ has been sleeping

Mach Definition 25

$ThreadStatistics$

$$ThreadExist$$
$$\underline{u}ser\_time : THREAD \nrightarrow \mathbb{N}$$
$$\underline{s}ystem\_time : THREAD \nrightarrow \mathbb{N}$$
$$\underline{c}pu\_time : THREAD \nrightarrow \mathbb{N}$$
$$\underline{s}leep\_time : THREAD \nrightarrow \mathbb{N}$$

$$\operatorname{dom} \underline{u}ser\_time = \operatorname{dom} \underline{s}ystem\_time = \operatorname{dom} \underline{c}pu\_time = \operatorname{dom} \underline{s}leep\_time = \underline{t}hread\_exists$$

Review Note:
Should the domain of $\underline{s}leep\_time$ be all threads or only those with a particular run state?

4.2.9 Machine State

The system records the machine state of each thread. Typically, the structure of the machine state varies depending upon the architecture of the machine to which the thread is assigned. The type $SUPP\_MACHINE\_ARCH$ represents the set of supported machine architectures. The set $THREAD\_STATE\_INFO\_TYPES$ denotes the names of the various structures that are associated with the supported architectures. The type $THREAD\_STATE\_INFO$ denotes the possible values of the state information recorded for a thread.

The expression $State\_info\_avail(arch)$ denotes the types of state information which the architecture supports.

Mach Definition 26

$$[SUPP\_MACHINE\_ARCH]$$
$$[THREAD\_STATE\_INFO\_TYPES, THREAD\_STATE\_INFO]$$

$$State\_info\_avail : SUPP\_MACHINE\_ARCH \longrightarrow \mathbb{P}\ THREAD\_STATE\_INFO\_TYPES$$

The expression $\underline{t}hread\_state(thread, info\_type)$ returns the indicated type of state information recorded for $thread$.

Mach Definition 27

$ThreadMachineState$

$$ThreadExist$$
$$\underline{t}hread\_state : THREAD \times THREAD\_STATE\_INFO\_TYPES \nrightarrow THREAD\_STATE\_INFO$$

$$\operatorname{dom} \underline{t}hread\_state = \underline{t}hread\_exists \times THREAD\_STATE\_INFO\_TYPES$$

Review Note:
Actually, the current instruction pointer is part of the machine state rather than being a separate state component.

Mach Definition 28

$$
\begin{array}{l}
\textit{Threads} \\
\hline
TasksAndThreads \\
ThreadPri \\
ThreadSchedPolicy \\
ThreadInstruction \\
ThreadExecStatus \\
ThreadStatistics \\
ThreadMachineState \\
ThreadSampling \\
TaskSampling
\end{array}
$$

4.3 Port Name Space

Each task uses its own (local) set of names to refer to ports. The set $NAME$ is used to name ports in a task's name space.

Mach Definition 29

$$[NAME]$$

The names $Mach\_port\_null$ and $Mach\_port\_dead$ are reserved. They will never be used as an index in a task's port name space. The remainder of this section discusses the three types of entities that can be in name spaces: port rights, port sets, and dead names.

Mach Definition 30

$$
\begin{array}{l}
Mach\_port\_dead : NAME \\
Mach\_port\_null : NAME
\end{array}
$$

4.3.1 Port Rights

A port is only of use to a task if the task holds some kind of right to the port. The types of available rights are defined via the type $RIGHT$. A right for a port allows a task to either send or receive messages via that port. The task may have either a general right to send messages via a port or a one-time right to do so. Thus, the elements of type $RIGHT$ are: $Send$, $Receive$, and $Send\_once$.

A $Capability$ is the combination of a port and a right to do something with that port.

Strictly speaking, a task associates a name with a particular right to a port, not simply with the port. The set $\underline{p}ort\_right\_rel$ relates the ports to which a task has rights with their right types and their local names. More specifically, each element of $\underline{p}ort\_right\_rel$ is a tuple of the form $(task, port, name, right, i)$. Such a tuple is an element of $\underline{p}ort\_right\_rel$ only when $name$ denotes in $task$'s name space a right of type $right$ to $port$. The i-value is used to allow a task to accumulate multiple send rights under the same name. For send-once or receive rights, the value of i is always equal to 1. For convenience, the expression $named\_port(task, name)$ denotes the port associated with $name$ in $task$'s name space.

At most one task can receive messages from a port at any given time. The expression $receiver(port)$ denotes the task (if any) that is currently permitted to receive messages from $port$, and $receiver\_name(port)$ denotes the receiver task's name for the port.

Many tasks may have $Send$ or $Send\_once$ rights to a port. The relation $sender$ indicates the tasks currently permitted to send messages to a port; an element $(port, task)$ is in $sender$ exactly when $task$ has a send right to $port$.

Mach Definition 31

$$RIGHT ::= Send \mid Receive \mid Send\_once$$

Capability

$port : PORT$
$right : RIGHT$

TasksAndPorts

$TaskExist$
$PortExist$
$\underline{p}ort\_right\_rel : \mathbb{P}(TASK \times PORT \times NAME \times RIGHT \times \mathbb{N}_1)$
$named\_port : TASK \times NAME \leftrightarrow PORT$
$receiver : PORT \leftrightarrow TASK$
$receiver\_name : PORT \leftrightarrow NAME$
$sender : PORT \leftrightarrow TASK$

$\underline{p}ort\_right\_rel \subseteq \underline{t}ask\_exists \times \underline{p}ort\_exists \times NAME \times RIGHT \times \mathbb{N}_1$
$\forall\, task : TASK;\ port : PORT;\ right : RIGHT;\ i : \mathbb{N}_1$
$\bullet\ (task, port, Mach\_port\_null, right, i) \notin \underline{p}ort\_right\_rel$
$\wedge\ (task, port, Mach\_port\_dead, right, i) \notin \underline{p}ort\_right\_rel$
$named\_port = \{\, task : TASK;\ port : PORT;\ name : NAME;\ right : RIGHT;\ i : \mathbb{N}_1$
$\quad \mid (task, port, name, right, i) \in \underline{p}ort\_right\_rel \bullet ((task, name), port)\}$
$receiver = \{\, task : TASK;\ port : PORT;\ name : NAME$
$\quad \mid (task, port, name, Receive, 1) \in \underline{p}ort\_right\_rel \bullet (port, task)\,\}$
$receiver\_name = \{\, task : TASK;\ port : PORT;\ name : NAME$
$\quad \mid (task, port, name, Receive, 1) \in \underline{p}ort\_right\_rel \bullet (port, name)\,\}$
$sender = \{\, task : TASK;\ port : PORT;\ name : NAME;\ right : RIGHT;\ i : \mathbb{N}_1$
$\quad \mid (((task, port, name, right, i) \in \underline{p}ort\_right\_rel) \wedge right \in \{Send, Send\_once\})$
$\quad \bullet\ (port, task)\,\}$

The i-value is called the user reference count. As noted above, it is equal to 1 for receive and send-once rights, but is of interest for send rights. The expression $s\_right\_ref\_count(task, name)$ returns the user reference count for $name$ in $task$'s name space (when it is a send right). There is a system-wide maximum number of references to a given send right which a task may accumulate, represented by Max_right_refs.

Mach Definition 32

$$Max\_right\_refs : \mathbb{N}_1$$

Mach Definition 33

$$
\begin{array}{l}
\textit{UserReferenceCount} \\
\hline
\textit{TasksAndPorts} \\
s\_right\_ref\_count : TASK \times NAME \pfun \mathbb{N}_1 \\
\hline
\forall task : TASK; port : PORT; name : NAME; right : \{Receive, Send\_once\}; i : \mathbb{N}_1 \\
\bullet (task, port, name, right, i) \in \underline{p}ort\_right\_rel \Rightarrow i = 1 \\
s\_right\_ref\_count = \{ task : \overline{T}ASK;\ port : PORT;\ name : NAME; i : \mathbb{N}_1 \\
\mid (task, port, name, Send, i) \in \underline{p}ort\_right\_rel \bullet ((task, name), i)\} \\
\forall task : TASK; name : NAM\overline{E} \bullet s\_right\_ref\_count(task, name) \leq Max\_right\_refs
\end{array}
$$

For convenience:

- The relations s_right, r_right, and so_right are used to identify the names of each of the types of rights which are associated with a given task. For example, $(task, name)$ is an element of s_right exactly when $name$ is a send right in $task$'s name space.
- The relation s_r_right is used to identify names that are either a receive or a send right.
- The relation $port\_right\_namep$ identifies names that are either receive, send, or send-once rights.

The semantics of Mach are such that send and receive rights within a task coalesce into a single name. In other words:

- If $name$ is a receive right for $port$ in $task$'s name space, then no other name in $task$'s name space may be a send right for $port$; the send rights must be associated with $name$, too.
- If $name$ is a send right for $port$ in $task$'s name space, then all of the send rights for $port$ in $task$'s name space are associated with $name$.

Note, however, that the same task can have multiple names associated with send-once rights for the same port. Mach prohibits a name that is a send or a receive right from also being a send-once right.

A message may be forcibly enqueued using a send right. In this case it will be added to the message queue of the named port even if the queue has reached its designated size limit. At most one message may be forcibly enqueued at a time using any given send right. After that message is removed from the queue, a message-accepted notification is sent and the send right can again be used to forcibly enqueue a message. The component $forcibly\_queued(task, name)$ denotes the message, if any, forcibly enqueued using a send right $name$ in $task$'s ipc name space.

Mach Definition 34

TasksAndRights

$$MessageExist$$
$$TasksAndPorts$$
$$s\_right : TASK \leftrightarrow NAME$$
$$r\_right : TASK \leftrightarrow NAME$$
$$so\_right : TASK \leftrightarrow NAME$$
$$s\_r\_right : TASK \leftrightarrow NAME$$
$$port\_right\_namep : TASK \leftrightarrow NAME$$
$$\underline{f}orcibly\_queued : (TASK \times NAME) \pfun MESSAGE$$

$$s\_right = \{\, task : TASK;\ port : PORT;\ name : NAME;\ i : \mathbb{N}_1 \mid (task, port, name, Send, i) \in \underline{p}ort\_right\_rel \bullet (task, name)\}$$
$$r\_right = \{\, task : TASK;\ port : PORT;\ name : NAME \mid (task, port, name, Receive, 1) \in \underline{p}ort\_right\_rel \bullet (task, name)\}$$
$$so\_right = \{\, task : TASK;\ port : PORT;\ name : NAME \mid (task, port, name, Send\_once, 1) \in \underline{p}ort\_right\_rel \bullet (task, name)\}$$
$$s\_r\_right = s\_right \cup r\_right$$
$$port\_right\_namep = s\_r\_right \cup so\_right$$

$$\operatorname{dom} \underline{f}orcibly\_queued \subseteq s\_right$$
$$\operatorname{ran} \underline{f}orcibly\_queued \subseteq \underline{m}essage\_exists$$

$$\mathsf{disjoint}\ \langle so\_right, s\_r\_right \rangle$$
$$\forall\, task : TASK;\ name_1, name_2 : NAME$$
$$\bullet\ (task, name_1) \in s\_r\_right \land (task, name_2) \in s\_r\_right$$
$$\land\ named\_port(task, name_1) = named\_port(task, name_2)$$
$$\Rightarrow name_1 = name_2$$

Review Note:
I'd like to tie the message indicated by $\underline{f}orcibly\_queued$ back to the port indicated by the send right, but I'm not sure this will be accurate.

4.3.2 Port Sets

A port set is a set of ports associated with a particular task and name. A port set is used to allow the receiving of a message via any member of the port set. Given a task and a port set name, the expression $port\_set(task, name)$ denotes the port set. The relation $port\_set\_namep$ identifies the port set names associated with each task. $containing\_set(port)$ denotes the name of the port set containing $port$, if any. Note that a port can be in at most one port set.

Mach prohibits the reserved names $Mach\_port\_null$ and $Mach\_port\_dead$ from being port set names or the inclusion of the same receive right in two different port sets.

Mach Definition 35

$$
\begin{array}{l}
\textbf{PortSets} \\
TaskExist \\
TasksAndRights \\
\underline{p}ort\_set\_rel : \mathbb{P}(TASK \times NAME \times \mathbb{P}\, PORT) \\
port\_set : (TASK \times NAME) \rightarrow\!\!\!\!\!+\!\!\!\!\!\rightarrow \mathbb{P}\, PORT \\
port\_set\_namep : TASK \leftrightarrow NAME \\
containing\_set : PORT \rightarrow\!\!\!\!\!+\!\!\!\!\!\rightarrow NAME \\
\hline
port\_set = \{task : TASK; name : NAME; set\_of\_ports : \mathbb{P}\, PORT \\
\quad \mid (task, name, set\_of\_ports) \in \underline{p}ort\_set\_rel \bullet ((task, name), set\_of\_ports)\} \\
port\_set\_namep = \mathrm{dom}\, port\_set \\
containing\_set = \{task : TASK; name : NAME; port : PORT \\
\quad \mid (task, name) \in port\_set\_namep \land port \in port\_set(task, name) \\
\quad \bullet (port, name)\} \\
\\
\mathrm{dom}\, port\_set\_namep \subseteq \underline{t}ask\_exists \\
\forall\, task : TASK; name : NAME; port : PORT \mid (task, name) \in \mathrm{dom}\, port\_set \\
\bullet\, port \in port\_set(task, name) \Rightarrow task = receiver(port) \\
\forall\, task : TASK; set\_of\_ports : \mathbb{P}\, PORT \\
\bullet\, ((task, Mach\_port\_null), set\_of\_ports) \notin port\_set \\
\land ((task, Mach\_port\_dead), set\_of\_ports) \notin port\_set \\
\forall\, task : TASK; name_1, name_2 : NAME \\
\mid (task, name_1) \in \mathrm{dom}\, port\_set \land (task, name_2) \in \mathrm{dom}\, port\_set \\
\bullet\, name_1 \neq name_2 \Rightarrow \mathsf{disjoint}\, \langle port\_set(task, name_1), port\_set(task, name_2)\rangle
\end{array}
$$

4.3.3 Dead Rights

A dead name is a name which previously named a send, receive, or send-once right for a task, but no longer does.[5] Each dead name in a task can have an associated count that is analogous to the reference count associated with send rights. This count is initially set based on the user reference counts for the right previously bearing the name. The count may be modified by subsequent actions of the kernel. The relation $\underline{d}ead\_right\_rel$ identifies the dead names and their associated counts for each task; an element $(task, name, i)$ is an element of $\underline{d}ead\_right\_rel$ if $name$ is a dead name in $task$ with associated count i. The previously defined constant, Max_right_refs, is a system-wide maximum for the reference count of a given dead right. For convenience:

- The relation $dead\_namep$ identifies the dead names associated with each task.
- The expression $dead\_right\_ref\_count(task, name)$ denotes the count associated with $name$ in $task$ (when $name$ is a dead name).

Mach prohibits $Mach\_port\_null$ and $Mach\_port\_dead$ from being dead names.

Mach Definition 36

[5] A dead name may also be specified in the body of a message in place of an actual port right.

$$
\begin{array}{l}
\textit{DeadRights} \\
\hline
\underline{d}ead\_right\_rel : \mathbb{P}(TASK \times NAME \times \mathbb{N}_1) \\
dead\_right\_ref\_count : TASK \times NAME \leftrightarrow \mathbb{N}_1 \\
dead\_namep : TASK \leftrightarrow NAME \\
\hline
dead\_right\_ref\_count = \{task : TASK; name : NAME; i : \mathbb{N}_1 \\
\mid (task, name, i) \in \underline{d}ead\_right\_rel \bullet ((task, name), i)\} \\
dead\_namep = \operatorname{dom} dead\_right\_ref\_count \\
\forall task : TASK; name : NAME \\
\bullet dead\_right\_ref\_count(task, name) \leq Max\_right\_refs \\
\forall task : TASK \\
\bullet (task, Mach\_port\_null) \notin dead\_namep \\
\wedge (task, Mach\_port\_dead) \notin dead\_namep
\end{array}
$$

4.3.4 Summary

A task's port right names (send, receive, and send-once), port set names, and dead names are mutually disjoint. The union of $port\_right\_namep$, $port\_set\_namep$, and $dead\_namep$ identifies the names in each task's name space. For convenience:

- The relation $local\_namep$ is used to denote this union.
- The expression $number\_of\_rights(task)$ is used to denote the number of names that $local\_namep$ associates with $task$. This is the current size of $task$'s name space.

Mach Definition 37

$$
\begin{array}{l}
\textit{PortNameSpace} \\
\hline
TaskExist \\
TasksAndPorts \\
TasksAndRights \\
UserReferenceCount \\
PortSets \\
DeadRights \\
local\_namep : TASK \leftrightarrow NAME \\
number\_of\_rights : TASK \leftrightarrow \mathbb{N} \\
\hline
\mathsf{disjoint}\ \langle port\_right\_namep, port\_set\_namep, dead\_namep \rangle \\
local\_namep = port\_right\_namep \cup port\_set\_namep \cup dead\_namep \\
\operatorname{dom} number\_of\_rights = \underline{t}ask\_exists \\
\forall task : TASK \mid task \in \underline{t}ask\_exists \\
\bullet number\_of\_rights(task) = \#(local\_namep (\!| \{ task \} |\!))
\end{array}
$$

4.4 Ports

This section describes data structures associated with ports.

4.4.1 Make Send Count

Each time the receiver for a port creates a new send right for the port, the system increments a counter associated with the port. The expression $\underline{m}ake\_send\_count(port)$ denotes the value of the counter associated with $port$. Note that this count does not necessarily represent the current number of send rights for the port since tasks other than the receiver can create send rights. Furthermore, the count does not necessarily represent the number of send rights the receiver has created because the count can directly be set to arbitrary values by user threads.

Mach Definition 38

$$
\begin{array}{l}
\textit{SendRightsCount} \\
\hline
\textit{PortExist} \\
\underline{m}ake\_send\_count : PORT \leftrightarrow \mathbb{N} \\
\hline
\mathrm{dom}\ \underline{m}ake\_send\_count = \underline{p}ort\_exists
\end{array}
$$

4.4.2 Message Queues

Each port has an associated message queue. A message queue can be thought of as a sequence of messages. In Mach, a task may set a limit on the number of messages that are permitted in a given message queue. The value $Mach\_port\_q\_limit\_default$ represents the default limit the kernel uses for newly allocated ports. The value $Mach\_port\_q\_limit\_max$ represents a system-imposed limit on the value a task may specify as the limit for a message queue.

Mach Definition 39

$$
\begin{array}{l}
Mach\_port\_q\_limit\_max : \mathbb{N} \\
Mach\_port\_q\_limit\_default : \mathbb{N}
\end{array}
$$

For each port, $\underline{q}\_limit(port)$ indicates the current limit set for the port. This denotes an intended bound on the number of messages in the associated message queue. The expression $port\_size(port)$ indicates the number of messages that are actually present in $port$'s message queue. Although it is intended that $port\_size(port)$ is always less than or equal to $\underline{q}\_limit(port)$, the kernel does not actually guarantee that this property always holds. Examples of ways in which the property may be violated include:

- The intended bound on the number of messages in a queue can be decreased below the number of messages already in the queue.
- Messages sent with a send-once right are delivered regardless of whether the destination port's queue is already full.
- Each name for a send right to a port may be used to forcibly enqueue one message at a time to the named full port.

The expression $\underline{m}essage\_in\_port\_rel(port)$ denotes the sequence of messages in the queue associated with $port$. Each message is contained in at most one message queue. For convenience, the expression $containing\_port(message)$ is used to indicate the port associated with the message queue to which $message$ belongs.

Each port has an associated sequence number that is used to properly sequence messages received through the port. The expression $\underline{s}equence\_no(port)$ indicates $port$'s current sequence number.

Mach Definition 40

$$
\begin{array}{l}
\textit{MessageQueues} \\
\hline
\textit{PortExist} \\
\underline{q}\_limit : PORT \mathrel{+\!\!\!\!\to} \mathbb{N} \\
\underline{m}essage\_in\_port\_rel : PORT \longrightarrow \operatorname{iseq} MESSAGE \\
port\_size : PORT \mathrel{+\!\!\!\!\to} \mathbb{N} \\
containing\_port : MESSAGE \mathrel{+\!\!\!\!\to} PORT \\
\underline{s}equence\_no : PORT \mathrel{+\!\!\!\!\to} \mathbb{Z} \\
\hline
containing\_port = \{\, message : MESSAGE;\ port : PORT \\
\quad \mid message \in \operatorname{ran}(\underline{m}essage\_in\_port\_rel(port)) \bullet message \mapsto port \,\} \\
(\forall\, port : \underline{p}ort\_exists \\
\bullet\ port\_size(port) = \#\,(\underline{m}essage\_in\_port\_rel(port)) \\
\quad \wedge\ \underline{q}\_limit(port) \leq Mach\_port\_q\_limit\_max) \\
\operatorname{dom} \underline{q}\_limit = \underline{p}ort\_exists \\
\operatorname{dom} \underline{m}essage\_in\_port\_rel = \underline{p}ort\_exists \\
\operatorname{dom} \underline{s}equence\_no = \underline{p}ort\_exists \\
\hline
\end{array}
$$

4.4.3 Summary

The data structures defined in this section consist of make-send counts, message queues, and sequence numbers associated with ports.

Mach Definition 41

$$
\begin{array}{l}
\textit{PortSummary} \\
\hline
\textit{SendRightsCount} \\
\textit{MessageQueues} \\
\hline
\end{array}
$$

4.5 Notifications

A task may request that a notification message be sent when one of the following changes occurs in the status of a port:

- The port is destroyed.
- The last send right for the port is deallocated.

A task may also request a notification message be sent when a send right becomes a dead name. In each case, the task requesting the notification must register a port to which the notification should be sent.

The relation $\underline{p}ort\_notify\_destroyed\_rel$ identifies the ports for which a destroyed notification has been requested and the associated notification ports. For convenience, $port\_notify\_destroyed(port)$ is used to denote the notification port registered for a destroyed notification on $port$.

The relation $\underline{p}ort\_notify\_no\_more\_senders\_rel$ identifies the ports for which a no-more-senders notification has been requested and the associated notification ports. For convenience, $port\_notify\_no\_more\_senders(port)$ is used to denote the notification port registered for a no-more-senders notification on $port$.

The relation $\underline{p}ort\_notify\_dead\_rel$ identifies the task-name pairs for which a dead-name notification has been requested and the associated notification ports. For convenience,

$port\_notify\_dead(task, name)$ is used to denote the notification port registered for a dead-name notification on $name$ in $task$'s name space.

The registered notification ports remain in force as long as both the port in question and the registered port exist regardless of whether the same tasks remain related to these ports.

Mach Definition 42

$$
\begin{array}{l}
\textbf{Notifications} \\
PortExist \\
TasksAndPorts \\
\underline{p}ort\_notify\_destroyed\_rel : PORT \leftrightarrow PORT \\
\underline{p}ort\_notify\_no\_more\_senders\_rel : PORT \leftrightarrow PORT \\
\underline{p}ort\_notify\_dead\_rel : \mathbb{P}(PORT \times TASK \times NAME) \\
port\_notify\_destroyed : PORT \nrightarrow PORT \\
port\_notify\_no\_more\_senders : PORT \nrightarrow PORT \\
port\_notify\_dead : TASK \times NAME \nrightarrow PORT \\
\hline
port\_notify\_destroyed = \underline{p}ort\_notify\_destroyed\_rel \\
port\_notify\_no\_more\_senders = \underline{p}ort\_notify\_no\_more\_senders\_rel \\
\forall\, task : TASK; port : PORT; name : NAME \\
\bullet\ ((port, task, name) \in \underline{p}ort\_notify\_dead\_rel \\
\Leftrightarrow ((task, name), port) \in port\_notify\_dead) \\
\\
\mathrm{dom}\, port\_notify\_destroyed = \underline{p}ort\_exists \\
\mathrm{dom}\, port\_notify\_no\_more\_senders = \underline{p}ort\_exists \\
\mathrm{dom}\, port\_notify\_dead = \mathrm{dom}\, named\_port \\
\\
\mathrm{ran}\, port\_notify\_destroyed \subseteq \underline{p}ort\_exists \cup \{Ip\_null\} \\
\mathrm{ran}\, port\_notify\_dead \subseteq \underline{p}ort\_exists \cup \{Ip\_null\} \\
\mathrm{ran}\, port\_notify\_no\_more\_senders \subseteq \underline{p}ort\_exists \cup \{Ip\_null\}
\end{array}
$$

Review Note:
Should the range of these functions also include Ip_dead? It seems that it should because the port could die. Should look at the code to see what happens if we try to send a notification in this situation.

4.6 Special Ports

This section describes the special ports known to the kernel. Each of the special ports is associated with some kernel entity.

4.6.1 Task Ports

In addition to the ports referenced in its port name space, each task has four special ports. The self port is used to request the kernel to perform actions upon the task. Any task holding a send right to a second task may use that right to request operations on the second task. The kernel is always the receiver for a task's self port. A task's sself port is normally equal to its self port, but may refer to a different port and have a task other than the kernel, such as a debugger, as its receiver. The relations $\underline{t}ask\_self\_rel$ and $\underline{t}ask\_sself\_rel$ identify the self and sself ports associated with each task.

The other two special ports are the exception port and the bootstrap port. A task receives exception messages from the kernel via its exception port. A task's bootstrap port is provided as a start-up means for a task to obtain a send right to a service port for a server that can provide the task start-up information. The relations $\underline{t}ask\_eport\_rel$ and $\underline{t}ask\_bport\_rel$ identify the exception port and bootstrap port associated with each task. The sself, exception and bootstrap ports may be modified. Unlike the self port, they may become lp_null or lp_dead.

For convenience:

- The expression $task\_self(task)$ denotes $task$'s self port.
- The expression $task\_sself(task)$ denotes $task$'s sself port.
- The expression $task\_eport(task)$ denotes $task$'s exception port.
- The expression $task\_bport(task)$ denotes $task$'s bootstrap port.
- The expression $self\_task(port)$ denotes the task (if any) having $port$ as its self port.

Mach Definition 43

$$
\begin{array}{l}
\textit{SpecialTaskPorts} \\
\hline
TaskExist \\
PortExist \\
Kernel \\
TasksAndPorts \\
\underline{t}ask\_self\_rel : TASK \leftrightarrow PORT \\
\underline{t}ask\_sself\_rel : TASK \leftrightarrow PORT \\
\underline{t}ask\_eport\_rel : TASK \leftrightarrow PORT \\
\underline{t}ask\_bport\_rel : TASK \leftrightarrow PORT \\
task\_self : TASK \rightarrowtail\!\!\!\!\!+ PORT \\
task\_sself : TASK \nrightarrow PORT \\
task\_eport : TASK \nrightarrow PORT \\
task\_bport : TASK \nrightarrow PORT \\
self\_task : PORT \rightarrowtail\!\!\!\!\!+ TASK \\
\hline
task\_self = \underline{t}ask\_self\_rel \\
task\_eport = \underline{t}ask\_eport\_rel \\
task\_bport = \underline{t}ask\_bport\_rel \\
task\_sself = \underline{t}ask\_sself\_rel \\
\mathrm{dom}\, task\_self = \mathrm{dom}\, task\_sself = \mathrm{dom}\, task\_eport = \mathrm{dom}\, task\_bport = \underline{t}ask\_exists \\
\mathrm{ran}\, task\_self \subset \underline{p}ort\_exists \\
\mathrm{ran}\, task\_sself \subset port\_pointer \\
\mathrm{ran}\, task\_eport \subset port\_pointer \\
\mathrm{ran}\, task\_bport \subset port\_pointer \\
self\_task = \underline{p}ort\_exists \lhd (task\_self^{\sim}) \\
\forall\, task : TASK \mid task \in \underline{t}ask\_exists \bullet receiver(task\_self(task)) = \underline{k}ernel
\end{array}
$$

4.6.2 Thread Ports

Each thread has a self port, sself port, and an exception port with purposes parallel to the corresponding special ports for tasks. The relations and functions $\underline{t}hread\_self\_rel$, $\underline{t}hread\_sself\_rel$, $\underline{t}hread\_eport\_rel$, $thread\_self$, $thread\_sself$, $thread\_eport$, and $self\_thread$ are used to denote these state components.

Mach Definition 44

$$
\begin{array}{l}
\textbf{SpecialThreadPorts} \\
\hline
ThreadExist \\
PortExist \\
TasksAndPorts \\
Kernel \\
\underline{t}hread\_self\_rel : THREAD \leftrightarrow PORT \\
\underline{t}hread\_sself\_rel : THREAD \leftrightarrow PORT \\
\underline{t}hread\_eport\_rel : THREAD \leftrightarrow PORT \\
thread\_self : THREAD \rightarrowtail\!\!\!\!\!\!+\; PORT \\
thread\_sself : THREAD \nrightarrow PORT \\
thread\_eport : THREAD \nrightarrow PORT \\
self\_thread : PORT \rightarrowtail\!\!\!\!\!\!+\; THREAD \\
\hline
\underline{t}hread\_self\_rel = thread\_self \\
\underline{t}hread\_sself\_rel = thread\_sself \\
\underline{t}hread\_eport\_rel = thread\_eport \\
\mathrm{dom}\, thread\_self = \underline{t}hread\_exists \\
\mathrm{dom}\, thread\_sself = \underline{t}hread\_exists \\
\mathrm{dom}\, thread\_eport = \underline{t}hread\_exists \\
\mathrm{ran}\, thread\_self \subset \underline{p}ort\_exists \\
\mathrm{ran}\, thread\_sself \subset port\_pointer \\
\mathrm{ran}\, thread\_eport \subset port\_pointer \\
self\_thread = \underline{p}ort\_exists \lhd (thread\_self^{\sim}) \\
\forall\, thread : THREAD \mid thread \in \underline{t}hread\_exists \bullet receiver(thread\_self(thread)) = \underline{k}ernel
\end{array}
$$

4.6.3 Memory Ports

A kernel and a memory object interact by engaging in a dialogue. The kernel sends messages to an object port and the object manager sends messages to a control port. There is also a name port used to identify the object in **vm_region** requests. The relations $\underline{o}bject\_port\_rel$, $\underline{c}ontrol\_port\_rel$, and $\underline{n}ame\_port\_rel$ are used to represent the binding between a memory and its associated ports. For a particular Mach host kernel, there is at most one of each type of port associated with a given memory. Furthermore, no object port is associated with more than one memory object. For convenience:

- The expressions $object\_port(memory)$, $control\_port(memory)$, and $name\_port(memory)$ are used to denote, respectively, the object, control, and name port for $memory$.
- The expression $object\_memory(port)$ denotes the memory object (if any) for which $port$ is the object port.
- The expression $control\_memory(port)$ denotes the memory object (if any) for which $port$ is the control port.

Memory objects are given a name port immediately upon allocation. However, they need not necessarily have object and control ports until a page that they back needs to be paged out.

Mach Definition 45

$$
\begin{array}{l}
\textbf{MemoriesAndPorts} \\
Kernel \\
MemoryExist \\
TasksAndPorts \\
\underline{o}bject\_port\_rel : MEMORY \leftrightarrow PORT \\
\underline{c}ontrol\_port\_rel : MEMORY \leftrightarrow PORT \\
\underline{n}ame\_port\_rel : MEMORY \leftrightarrow PORT \\
object\_port : MEMORY \rightarrowtail\!\!\!\!\!+\, PORT \\
control\_port : MEMORY \rightarrowtail\!\!\!\!\!+\, PORT \\
name\_port : MEMORY \rightarrowtail\!\!\!\!\!+\, PORT \\
object\_memory : PORT \rightarrowtail\!\!\!\!\!+\, MEMORY \\
control\_memory : PORT \rightarrowtail\!\!\!\!\!+\, MEMORY \\
\hline
\underline{o}bject\_port\_rel = object\_port \\
\underline{c}ontrol\_port\_rel = control\_port \\
\underline{n}ame\_port\_rel = name\_port \\
object\_port^{\sim} = object\_memory \\
control\_port^{\sim} = control\_memory \\
\mathrm{dom}\, \underline{c}ontrol\_port\_rel = \mathrm{dom}\, \underline{o}bject\_port\_rel \subseteq \mathrm{dom}\, \underline{n}ame\_port\_rel \\
\mathrm{dom}\, \underline{n}ame\_port\_rel = \underline{m}emory\_exists \\
\forall\, port : PORT \mid port \in \mathrm{ran}\, \underline{c}ontrol\_port\_rel \\
\bullet\ port \in \mathrm{dom}\, receiver \land receiver(port) = \underline{k}ernel \\
\forall\, port : PORT \mid port \in \mathrm{ran}\, \underline{n}ame\_port\_rel \\
\bullet\ port \in \mathrm{dom}\, receiver \land receiver(port) = \underline{k}ernel
\end{array}
$$

4.6.4 Host Ports

Each host has two associated ports: the control port and the name port. These ports are denoted by $\underline{h}ost\_control\_port$ and $\underline{h}ost\_name\_port$. The kernel is the receiver for each of these ports. The name port is used to service "unprivileged" requests while the control port is used to service "privileged" requests.

Mach Definition 46

$$
\begin{array}{l}
\textbf{HostsAndPorts} \\
Kernel \\
TasksAndPorts \\
\underline{h}ost\_control\_port : PORT \\
\underline{h}ost\_name\_port : PORT \\
\hline
(\underline{h}ost\_name\_port, \underline{k}ernel) \in receiver \\
(\underline{h}ost\_control\_port, \underline{k}ernel) \in receiver
\end{array}
$$

4.6.5 Processor Ports

Each processor has a port that is used to name it. The relation $\underline{p}rocessor\_port\_rel$ indicates the association between processors and their name ports. There is exactly one port associated with each processor. For convenience, $proc\_self(proc)$ and $the\_processor(port)$ are used to denote, respectively, the port associated with a given processor and the processor associated with a given port.

Each processor set has two associated ports: the control port and the name port. The relations $ps\_control\_port\_rel$ and $ps\_name\_port\_rel$ are used to represent the binding between a processor set and its associated ports. In Mach, there is exactly one of each type of port associated with each existing processor set. For convenience:

- The expression $controlled\_proc\_set(port)$ is used to indicate the processor set (if any) having $port$ as its control port.
- The expression $procset\_self(procset)$ is used to indicate $procset$'s control port.
- The expression $named\_proc\_set(port)$ is used to indicate the processor set (if any) having $port$ as its name port.
- The expression $procset\_name\_port(procset)$ is used to indicate $procset$'s name port.

Mach Definition 47

$$
\begin{array}{l}
\textbf{ProcessorsAndPorts} \\
\hline
Kernel \\
TasksAndPorts \\
processor\_port\_rel : PROCESSOR \leftrightarrow PORT \\
ps\_control\_port\_rel : PROCESSOR\_SET \leftrightarrow PORT \\
ps\_name\_port\_rel : PROCESSOR\_SET \leftrightarrow PORT \\
proc\_self : PROCESSOR \nrightarrow PORT \\
the\_processor : PORT \nrightarrow PROCESSOR \\
controlled\_proc\_set : PORT \nrightarrow PROCESSOR\_SET \\
procset\_self : PROCESSOR\_SET \nrightarrow PORT \\
named\_proc\_set : PORT \nrightarrow PROCESSOR\_SET \\
procset\_name\_port : PROCESSOR\_SET \nrightarrow PORT \\
\hline
\mathrm{dom}\, ps\_control\_port\_rel = \mathrm{dom}\, ps\_name\_port\_rel \\
processor\_port\_rel^{\sim} = the\_processor \\
processor\_port\_rel = proc\_self \\
ps\_control\_port\_rel^{\sim} = controlled\_proc\_set \\
ps\_control\_port\_rel = procset\_self \\
ps\_name\_port\_rel^{\sim} = named\_proc\_set \\
ps\_name\_port\_rel = procset\_name\_port \\
\forall port : PORT \mid port \in \mathrm{ran}\, ps\_control\_port\_rel \\
\bullet\ port \in \mathrm{dom}\, receiver \land receiver(port) = kernel \\
\forall port : PORT \mid port \in \mathrm{ran}\, ps\_name\_port\_rel \\
\bullet\ port \in \mathrm{dom}\, receiver \land receiver(port) = kernel \\
\forall port : PORT \mid port \in \mathrm{ran}\, processor\_port\_rel \\
\bullet\ port \in \mathrm{dom}\, receiver \land receiver(port) = kernel \\
\hline
\end{array}
$$

4.6.6 Device Ports

Each device is represented by a unique port. The relation $device\_port\_rel$ identifies the device port representing each device. The kernel is the receiver for a device port. For convenience:

- The expression $device\_port(dev)$ is used to denote dev's device port.
- The expression $port\_device(port)$ is used to denote the device (if any) having $port$ as its device port.

Mach Definition 48

$$
\begin{array}{l}
\textbf{schema } DevicesAndPorts \\
\quad TasksAndPorts \\
\quad Kernel \\
\quad \underline{d}evice\_port\_rel : DEVICE \leftrightarrow PORT \\
\quad device\_port : DEVICE \rightarrowtail\!\!\!\!\!\!+ PORT \\
\quad port\_device : PORT \rightarrowtail\!\!\!\!\!\!+ DEVICE \\
\hline
\quad device\_port = \underline{d}evice\_port\_rel \\
\quad port\_device = \underline{d}evice\_port\_rel^{\sim} \\
\quad \forall port : PORT \mid port \in \operatorname{ran} \underline{d}evice\_port\_rel \\
\quad \bullet\ port \in \operatorname{dom} receiver \land receiver(port) = \underline{k}ernel
\end{array}
$$

4.6.7 Device Master Port

Tasks gain access to devices through the device master port which is denoted by $\underline{m}aster\_device\_port$. The kernel is the receiver for this port.

Mach Definition 49

$$
\begin{array}{l}
\textbf{schema } MasterDevicePort \\
\quad TasksAndPorts \\
\quad Kernel \\
\quad \underline{m}aster\_device\_port : PORT \\
\hline
\quad (\underline{m}aster\_device\_port, \underline{k}ernel) \in receiver
\end{array}
$$

4.6.8 Summary

Each special port for which the kernel is always the receiver must be distinct from all of the other special ports for which the kernel is always the receiver. For example, no two tasks may have the same self port, and no port may be both a task self port and a thread self port. Note, however, that the kernel does not prohibit overlaps between the special ports for which the kernel is always the receiver and the other special ports. For example, a task's bootstrap port might be set to some others task's self port (even though this would probably not serve any useful purpose).

Mach Definition 50

$$
\begin{array}{l}
\textbf{schema } SpecialPurposePorts \\
\quad SpecialTaskPorts \\
\quad SpecialThreadPorts \\
\quad MemoriesAndPorts \\
\quad HostsAndPorts \\
\quad ProcessorsAndPorts \\
\quad DevicesAndPorts \\
\quad MasterDevicePort \\
\hline
\quad \mathsf{disjoint}\ \langle \operatorname{ran} task\_self, \operatorname{ran} thread\_self, \operatorname{ran} control\_port, \operatorname{ran} object\_port, \\
\qquad \operatorname{ran} name\_port, \{\underline{h}ost\_control\_port\}, \{\underline{h}ost\_name\_port\}, \\
\qquad \operatorname{ran} \underline{p}s\_control\_port\_rel, \operatorname{ran} \underline{p}s\_name\_port\_rel, \operatorname{ran} \underline{p}rocessor\_port\_rel, \\
\qquad \operatorname{ran} \underline{d}evice\_port\_rel, \{\underline{m}aster\_device\_port\}\rangle
\end{array}
$$

Editorial Note:
The following needs some revision:

- Add port classes for pager name ports and pager (object) ports.
- Correct the misunderstanding that a port in a port class must have the kernel as the receiver. While this is true for most classes, memory object (pager) ports are a notable exception.

The type $PORT\_CLASS$ denotes the classes of ports for which the kernel is the receiver. These are Pc_task, Pc_thread, $Pc\_host\_control$, Pc_host_name, $Pc\_ps\_control$, Pc_ps_name, $Pc\_processor$, Pc_memory, and Pc_device.

If the kernel is the receiver for $port$, then the expression $\underline{p}ort\_class(port)$ denotes $port$'s class.

Mach Definition 51

$$PORT\_CLASS ::= Pc\_task \mid Pc\_thread \mid Pc\_host\_control \mid Pc\_host\_name \mid Pc\_ps\_control \mid Pc\_ps\_name \mid Pc\_processor \mid Pc\_memory \mid Pc\_device$$

$PortClasses$

$SpecialPurposePorts$
$\underline{p}ort\_class : PORT \leftrightarrow PORT\_CLASS$

$\forall port : PORT$
$\bullet (port \in \operatorname{ran} task\_self \Rightarrow (port, Pc\_task) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} thread\_self \Rightarrow (port, Pc\_thread) \in \underline{p}ort\_class)$
$\land (port = \underline{h}ost\_control\_port \Rightarrow (port, Pc\_host\_control) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} \underline{d}evice\_port\_rel \Rightarrow (port, Pc\_device) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} \underline{c}ontrol\_port\_rel \Rightarrow (port, Pc\_memory) \in \underline{p}ort\_class)$
$\land (port = \underline{h}ost\_name\_port \Rightarrow (port, Pc\_host\_name) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} \underline{p}s\_control\_port\_rel \Rightarrow (port, Pc\_ps\_control) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} \underline{p}s\_name\_port\_rel \Rightarrow (port, Pc\_ps\_name) \in \underline{p}ort\_class)$
$\land (port \in \operatorname{ran} \underline{p}rocessor\_port\_rel \Rightarrow (port, Pc\_processor) \in \underline{p}ort\_class)$

4.7 Total Send Rights

In addition to the send rights contained in the port name spaces associated with the tasks, the kernel maintains so-called naked send rights to the special ports. We occasionally need to know the total number of send rights to a given port including both those recorded in a name space and the naked rights. Naked rights are associated with the following ports: $task\_sself$, $task\_eport$, $task\_bport$, $thread\_sself$ and $thread\_eport$. We define $port\_right\_seq$ to be any sequence of the elements of the set $\underline{p}ort\_right\_rel$ (the precise ordering of elements is not important for our purposes). The expression $total\_name\_space\_srights(port)$ denotes the number of send rights to $port$ in all name spaces, and $total\_naked\_srights(port)$ denotes the total number of send rights to $port$ that are not stored in any name space. The expression $total\_srights(port)$ is the sum of these two numbers.

Review Note:

Need to determine if naked send rights are implied by any other special port relationships. Note that a naked send right is *not* created for the self port relationships (e.g., $thread\_self$).

Need to determine whether rights in messages count as naked send rights too.

Mach Definition 52

$$
\begin{array}{l}
\textit{TotalSendRights} \\
\hline
PortExist \\
TasksAndPorts \\
SpecialPurposePorts \\
port\_right\_seq : \mathrm{seq}(TASK \times PORT \times NAME \times RIGHT \times \mathbb{N}_1) \\
total\_name\_space\_srights : PORT \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{N} \\
total\_naked\_srights : PORT \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{N} \\
total\_srights : PORT \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{N} \\
\hline
\mathrm{ran}\ port\_right\_seq = \underline{p}ort\_right\_rel \\
\#port\_right\_seq = \#\underline{p}ort\_right\_rel \\
\\
(\forall\, port : PORT \mid port \in \underline{p}ort\_exists \\
\bullet\ total\_name\_space\_srights(port) \\
\qquad = Seq\_plus(squash\,\{task : TASK;\ name : NAME;\ i, n : \mathbb{N}_1 \\
\qquad\qquad \mid (n, (task, port, name, Send, i)) \in port\_right\_seq \\
\qquad\qquad \bullet\ (n, i)\}) \\
\wedge\ total\_naked\_srights(port) = \#(task\_sself \rhd \{port\}) \\
\qquad + \#(task\_eport \rhd \{port\}) \\
\qquad + \#(task\_bport \rhd \{port\}) \\
\qquad + \#(thread\_sself \rhd \{port\}) \\
\qquad + \#(thread\_eport \rhd \{port\}) \\
\wedge\ total\_srights(port) = total\_name\_space\_srights(port) + total\_naked\_srights(port)) \\
\hline
\end{array}
$$

4.8 Registered Rights

Each task has a finite array of send rights, intended to use for access to the Network Name Server, the Environment Manager, and the Service server (although they may have any use). These rights are called “registered,” to denote the fact that the kernel knows their identity. The expression $\underline{r}egistered\_rights(task)$ denotes the set of names of rights registered for $task$. There may be more than three registered rights, in fact their number need only be less than or equal to the system constant $Task\_port\_register\_max$. The kernel has three constants $Name\_server\_slot$, $Environment\_slot$, and $Service\_slot$ which tell it which element of the array refers to each of these servers.

Mach Definition 53

$$
\begin{array}{|l}
Task\_port\_register\_max : \mathbb{N} \\
Name\_server\_slot : \mathbb{N} \\
Environment\_slot : \mathbb{N} \\
Service\_slot : \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textbf{RegisteredRights} \\
\hline
TaskExist \\
\underline{r}egistered\_rights : TASK \rightarrowtail \operatorname{seq} PORT \\
\hline
\operatorname{dom} \underline{r}egistered\_rights = \underline{t}ask\_exists \\
\forall\, task : TASK \mid task \in \underline{t}ask\_exists \\
\bullet\ \#(\underline{r}egistered\_rights(task)) \leq Task\_port\_register\_max
\end{array}
$$

4.9 Memory System

This section describes the components of the Mach system that are used to provide tasks with address spaces.

4.9.1 Memory

Each memory can be viewed as mapping a memory offset to a value. Essentially, a memory can be viewed as an array of values indexed by offsets; the only difference is that a memory may have holes in the sense that some offsets do not map to any value. The mapping from offsets to values is defined by the memory's manager. As described in Section 4.9.2, the kernel becomes aware of pieces of this mapping as data is cached in resident pages. The types $OFFSET$ and $WORD$ denote, respectively, the sets of memory offsets and memory values.

The kernel maintains a copy strategy for each memory object. This strategy is one of the following:

- $Memory\_copy\_none$ —

Review Note:
We need to figure out the meaning of each strategy.

- $Memory\_copy\_call$ —
- $Memory\_copy\_delay$ —
- $Memory\_copy\_temporary$ —

These values comprise the elements of the type $MEMORY\_COPY\_STRATEGY$. The expression $\underline{c}opy\_strategy(memory)$ denotes the copy strategy recorded for $memory$.

The kernel cannot request access permissions and data from a memory object until it has received a **memory_object_ready** command (normally in reply to a **memory_object_init** request). The set $\underline{i}nitialized$ denotes the set of memory objects for which this has occurred.

The kernel records which memory objects may be cached; the set $\underline{m}ay\_cache$ denotes the set of such memory objects. The memory performance for a memory object is influenced by its copy strategy and whether it can be cached.

A memory can be either managed or unmanaged. The set $\underline{m}anaged$ denotes the set of memories that are managed. Corresponding to each such memory there is a task acting as the memory's manager. The manager for $memory$ is denoted by $\underline{m}anager(memory)$. Each memory having an object port is managed.

Similarly, memories can be temporary or non-temporary. The set $\underline{t}emporary\_rel$ denotes the set of memories that are temporary.

If the page of data corresponding to a given memory-offset pair is not resident when a thread attempts access, then the thread is blocked on a page fault. The expression $\underline{m}emory\_fault(memory, offset)$ indicates the set of threads that are currently blocked on a page fault generated by access to a given memory-offset pair.

Temporary memory is backed by the default memory manager. The kernel records a port identifying the current default memory manager. This port is denoted by $\underline{d}efault\_mem\_manager$.

A null value is used to indicate the lack of a memory filling a particular function in a virtual memory map entry.

Review Note:
Need to figure out how $\underline{d}efault\_mem\_manager$ relates to $\underline{m}anaged$ and $\underline{m}anager$.

Mach Definition 54

$$[WORD, OFFSET]$$

$$\begin{array}{l} MEMORY\_COPY\_STRATEGY ::= Memory\_copy\_none \mid Memory\_copy\_call \\ \quad \mid Memory\_copy\_delay \mid Memory\_copy\_temporary \end{array}$$

$$\begin{array}{|l} \textit{— Memory —} \\ MemoriesAndPorts \\ PortExist \\ \underline{c}opy\_strategy : MEMORY \mathrel{+\!\!\!\!\rightarrow} MEMORY\_COPY\_STRATEGY \\ \underline{i}nitialized : \mathbb{P}\ MEMORY \\ \underline{m}ay\_cache : \mathbb{P}\ MEMORY \\ \underline{m}anaged : \mathbb{P}\ MEMORY \\ \underline{m}anager : MEMORY \mathrel{+\!\!\!\!\rightarrow} TASK \\ \underline{t}emporary\_rel : \mathbb{P}\ MEMORY \\ \underline{m}emory\_fault : MEMORY \times OFFSET \mathrel{+\!\!\!\!\rightarrow} \mathbb{P}\ THREAD \\ \underline{d}efault\_mem\_manager : PORT \\ \hline \underline{d}efault\_mem\_manager \in \underline{p}ort\_exists \\ \underline{m}anaged = \mathrm{dom}\ object\_port \\ \mathrm{dom}\ object\_port \subseteq \mathrm{dom}\ \underline{m}anager \\ \underline{i}nitialized \subseteq \mathrm{dom}\ object\_port \\ \underline{m}ay\_cache \subseteq \underline{i}nitialized \\ \underline{i}nitialized = \mathrm{dom}\ \underline{c}opy\_strategy \\ \forall\ memory : MEMORY;\ offset : OFFSET \\ \mid (memory, offset) \in \mathrm{dom}\ \underline{m}emory\_fault \\ \quad \wedge\ \underline{m}emory\_fault(memory, offset) \neq \emptyset \\ \bullet\ memory \in \underline{m}anaged \end{array}$$

4.9.2 Pages

At the physical level, pages relate page offsets and values in much the same way as memories relate memory offsets and values. The relation $\underline{p}age\_word\_rel$ identifies the binding between page-offset pairs and words of data. Since at most one value can be stored at a given page offset, $\underline{p}age\_word\_rel$ is actually a function mapping page-offset pairs to values. For convenience,

$page\_word\_fun(page)(page\_offset)$ is used to denote the word of data at offset $page\_offset$ of page $page$.

Each page represents some area of memory. The relation $\underline{r}epresents\_rel$ indicates the binding between pages and memory-offset pairs. This relation should be interpreted as indicating the memory and offset within that memory of the beginning of the data that a page represents. Since each area of memory is represented by at most one page, the function $representing\_page$ denoting the page representing an area of memory can be defined. Each page in the range of this function represents some area of memory. For convenience:

- The set $represents\_memory$ is used to denote the set of pages that represent some area of memory.
- The set $represented$ is used to denote the set of memory-offset pairs that are represented by some page.
- The expressions $represented\_memory(page)$ and $represented\_offset(page)$ denote, respectively, the memory and offset that $page$ represents.

When a page is modified, it becomes dirty. The set $\underline{d}irty\_rel$ denotes the set of dirty pages. Upon evicting a page, the kernel checks whether the page is dirty. If it is, then the contents of the page are sent to the appropriate memory manager for it to record the updates. A memory manager may instruct the kernel that it will not retain a copy of a page that it has provided to the kernel by indicating that the page is precious. Whenever the kernel evicts a precious page, it sends the contents of the page to the appropriate memory manager regardless of whether the page is dirty. By instructing the kernel that a page is precious, a memory manager can relieve itself of the responsibility of retaining a copy of a page while the page is resident; the memory manager can rely on the kernel to inform it of the page's current contents whenever the page is evicted. The set $\underline{p}recious$ is used to denote the set of precious pages.

Mach Definition 55

$[PAGE\_OFFSET]$

$$
\begin{array}{l}
\textit{PageAndMemory} \\
\hline
\underline{p}age\_word\_rel : \mathbb{P}((PAGE \times PAGE\_OFFSET) \times WORD) \\
page\_word\_fun : PAGE \pfun PAGE\_OFFSET \longrightarrow WORD \\
\underline{r}epresents\_rel : PAGE \longleftrightarrow (MEMORY \times OFFSET) \\
representing\_page : MEMORY \times OFFSET \pfun PAGE \\
represents\_memory : \mathbb{P}\ PAGE \\
represented : \mathbb{P}(MEMORY \times OFFSET) \\
represented\_memory : PAGE \pfun MEMORY \\
represented\_offset : PAGE \pfun OFFSET \\
\underline{d}irty\_rel : \mathbb{P}\ PAGE \\
\underline{p}recious : \mathbb{P}\ PAGE \\
\hline
(\forall\ page : PAGE; page\_offset : PAGE\_OFFSET; word : WORD \\
\bullet\ page\_word\_fun(page)(page\_offset) = word \\
\Leftrightarrow ((page, page\_offset), word) \in \underline{p}age\_word\_rel) \\
represents\_memory \subseteq \operatorname{dom} page\_word\_fun \\
representing\_page = \underline{r}epresents\_rel^{\sim} \\
\underline{d}irty\_rel \subseteq represents\_memory = \operatorname{ran} representing\_page \\
represented = \operatorname{dom} representing\_page \\
represented\_memory = \{memory : MEMORY; offset : OFFSET; page : PAGE \\
\quad \mid (page, (memory, offset)) \in \underline{r}epresents\_rel \bullet (page, memory)\} \\
represented\_offset = \{memory : MEMORY; offset : OFFSET; page : PAGE \\
\quad \mid (page, (memory, offset)) \in \underline{r}epresents\_rel \bullet (page, offset)\} \\
\underline{p}recious \subseteq represents\_memory \\
\hline
\end{array}
$$

Mach allows pages to be locked against particular types of accesses. This is represented by associating a set of protections with each page. The protections are of type $PROTECTION$ which is comprised of the elements $Read$, $Write$, and $Execute$. The relation $\underline{p}age\_lock\_rel$ indicates the access modes against which a page is locked. For convenience $page\_locks(page)$ is defined to be the set of access modes against which $page$ is locked.

Mach Definition 56

$$PROTECTION ::= Read \mid Write \mid Execute$$

$$
\begin{array}{l}
\textit{Lock} \\
\hline
\underline{p}age\_lock\_rel : PAGE \longleftrightarrow \mathbb{P}\ PROTECTION \\
page\_locks : PAGE \pfun \mathbb{P}\ PROTECTION \\
\hline
\underline{p}age\_lock\_rel = page\_locks \\
\hline
\end{array}
$$

4.9.3 Address Space

The set $\underline{a}llocated$ is used to denote the set of $TASK$-$PAGE\_INDEX$ pairs that have been allocated in a task's address space. A task-index pair may be mapped to a memory area. Using the previously defined state components, these memory areas can be related to the physical pages used to contain the data when it is paged out. Thus, a task's address space completes the picture of mapping virtual addresses to physical pages and values. Note, however, that not all allocated addresses need be mapped to memory. The relation $\underline{m}ap\_rel$ associates task-index pairs with memory-offset pairs. There is at most one memory-offset pair associated with each task-index pair. For convenience:

- The expressions $mapped\_memory(task, index)$ and $mapped\_offset(task, index)$ are used to denote the memory and offset corresponding to a given task-index pair.
- The set $mapped$ is used to denote the set of memories to which some task-index pair maps.

Mach Definition 57

$[PAGE\_INDEX]$

$$
\begin{array}{l}
\textit{AddressSpace} \\
\hline
\underline{m}ap\_rel : (TASK \times PAGE\_INDEX) \leftrightarrow (MEMORY \times OFFSET) \\
mapped\_memory : TASK \times PAGE\_INDEX \nrightarrow MEMORY \\
mapped\_offset : TASK \times PAGE\_INDEX \nrightarrow OFFSET \\
\underline{a}llocated : \mathbb{P}(TASK \times PAGE\_INDEX) \\
mapped : \mathbb{P}\ MEMORY \\
\hline
\mathrm{dom}\ \underline{m}ap\_rel = \mathrm{dom}\ mapped\_memory = \mathrm{dom}\ mapped\_offset \\
\mathrm{dom}\ \underline{m}ap\_rel \subseteq \underline{a}llocated \\
mapped = \mathrm{ran}\ mapped\_memory \\
\forall\ task\_va\_pair : TASK \times PAGE\_INDEX;\ memory : MEMORY;\ offset : OFFSET \\
\bullet\ (task\_va\_pair, (memory, offset)) \in \underline{m}ap\_rel \\
\Leftrightarrow (mapped\_memory(task\_va\_pair) = memory \\
\qquad \wedge\ mapped\_offset(task\_va\_pair) = offset)
\end{array}
$$

4.9.4 Memory Protection

Mach protects memory objects by assigning protections to each page in a task's address space. Three sets of protections are associated with each page in a task's address space. The Mach protection holds currently applicable protection limits as indicated by users. The maximum protection limits the allowable values for the Mach protection. The third set, the current protections, is what actually limits a task's access to a page. This is a DTOS addition and will be further defined in Section 5.9.[6]

We use $\underline{m}ach\_protection$ to denote the relation between tasks, pages, and Mach protection sets. The pair $((task, page\_index), protection\_set)$ is an element of $\underline{m}ach\_protection$ if $protection\_set$ is the set of protections most recently established by a user request to set the Mach protections for $page\_index$. We model maximum protections similarly by defining $\underline{m}ax\_protection(task, page\_index)$ to denote the maximum protection that $task$ is permitted to the memory it has mapped at $page\_index$.

Mach Definition 58

$$
\begin{array}{l}
\textit{MachProtection} \\
\hline
\underline{m}ach\_protection : (TASK \times PAGE\_INDEX) \nrightarrow \mathbb{P}\ PROTECTION \\
\underline{m}ax\_protection : (TASK \times PAGE\_INDEX) \nrightarrow \mathbb{P}\ PROTECTION \\
\hline
\mathrm{dom}\ \underline{m}ach\_protection = \mathrm{dom}\ \underline{m}ax\_protection \\
\forall\ task\_page\_index : TASK \times PAGE\_INDEX \\
\mid\ task\_page\_index \in \mathrm{dom}\ \underline{m}ach\_protection \\
\bullet\ \underline{m}ach\_protection(task\_page\_index) \subseteq \underline{m}ax\_protection(task\_page\_index)
\end{array}
$$

[6] The Mach protection in DTOS is called the current protection in Mach and is used in Mach to control a task's access of pages. The terminology has been changed here to remain consistent with the prototype which must take into account the decisions of the security server when determining the current protections.

4.9.5 Memory Inheritance

For each memory region within a task's address space, Mach records an inheritance attribute that indicates the manner in which child tasks inherit the memory. The possible options are:

- $Inheritance\_option\_share$ — indicates the region should be shared by the parent and child
- $Inheritance\_option\_copy$ — indicates the region should be shared by the parent and child until one of them writes to the region; once a modification occurs, a copy-on-write is performed
- $Inheritance\_option\_none$ — indicates the region should not be made accessible to the child

These values comprise the elements of the type $INHERITANCE\_OPTION$.

The expression $\underline{i}nheritance(task, page\_index)$ indicate the inheritance option associated with the region indicated by $page\_index$ in $task$'s address space.

Mach Definition 59

$$INHERITANCE\_OPTION ::= Inheritance\_option\_share \mid Inheritance\_option\_copy \mid Inheritance\_option\_none$$

Inheritance

$$\underline{i}nheritance : TASK \times PAGE\_INDEX \mapsto INHERITANCE\_OPTION$$

4.9.6 Shadow Memories

A memory, $memory_1$, is said to back a second memory, $memory_2$, if $memory_1$'s manager takes responsibility for pages of $memory_2$ that are not handled by $memory_2$'s manager. The relation $\underline{b}acking\_rel$ indicates when $memory_1$ backs $memory_2$ at a given offset within $memory_1$. Each memory is backed by at most one memory-offset pair. Furthermore, a memory may back at most one other memory. For convenience, $backing\_memory(memory)$ and $backing\_offset(memory)$ are used to denote, respectively, the memory and offset backing $memory$.

Whenever $memory_1$ backs $memory_2$, $memory_2$ is said to shadow $memory_1$. For convenience:

- The expression $shadow\_memories(memory)$ indicates the singleton set of memories backed by $memory$. $shadow\_memories$ is defined only for those memories that back another memory.
- The expression $backing\_chain(memory)$ indicates the sequence of memories backing $memory$.

If a memory is not backed by any memories, then its backing chain is empty. If $memory_1$ is backed by $memory_2$ then the backing chain for $memory_1$ consists of $memory_2$ followed by the backing chain for $memory_2$. For example, suppose $memory_2$ backs $memory_1$, $memory_3$ backs $memory_2$, and no memory backs $memory_3$. Then, the backing chains for $memory_3$, $memory_2$, and $memory_1$ are, respectively, $\langle\rangle$, $\langle memory_3\rangle$, and $\langle memory_2, memory_3\rangle$. Mach does not permit cycles to occur in the sequence of memories backing a memory. Thus, we require that no memory be present in its backing chain.

Mach Definition 60

ShadowMemories

$$\underline{b}acking\_rel : \mathbb{P}(MEMORY \times MEMORY \times OFFSET)$$
$$backing\_memory : MEMORY \rightarrowtail\!\!\!\!\!+ MEMORY$$
$$backing\_offset : MEMORY \rightarrowtail\!\!\!\!\!+ OFFSET$$
$$shadow\_memories : MEMORY \longrightarrow \mathbb{P}\, MEMORY$$
$$backing\_chain : MEMORY \longrightarrow \mathrm{seq}\, MEMORY$$

$$\forall\, memory_1, memory_2 : MEMORY;\ offset : OFFSET$$
$$\bullet\ (memory_1, memory_2, offset) \in \underline{b}acking\_rel$$
$$\Leftrightarrow ((memory_2, memory_1) \in backing\_memory$$
$$\quad \wedge (memory_2, offset) \in backing\_offset)$$
$$\mathrm{dom}\, shadow\_memories = \mathrm{ran}\, backing\_memory$$
$$\forall\, memory_1 : MEMORY \mid memory_1 \in \mathrm{dom}\, shadow\_memories$$
$$\bullet\ shadow\_memories(memory_1)$$
$$= \{memory_2 : MEMORY$$
$$\quad \mid (\exists\, offset : OFFSET \bullet (memory_1, memory_2, offset) \in \underline{b}acking\_rel)\}$$
$$\forall\, memory_1 : MEMORY \mid memory_1 \in \mathrm{dom}\, shadow\_memories$$
$$\bullet\ \#(shadow\_memories(memory_1)) = 1$$
$$\forall\, memory : MEMORY$$
$$\bullet\ memory \notin \mathrm{dom}\, backing\_memory \Rightarrow \#(backing\_chain(memory)) = 0$$
$$\wedge (memory \in \mathrm{dom}\, backing\_memory$$
$$\quad \Rightarrow backing\_chain(memory)$$
$$\qquad = \langle backing\_memory(memory)\rangle$$
$$\qquad\quad {}^\frown backing\_chain(backing\_memory(memory)))$$
$$\forall\, memory : MEMORY \bullet memory \notin \mathrm{ran}(backing\_chain(memory))$$

4.9.7 Page Wiring

To prevent critical pages from being evicted, Mach allows tasks to wire pages. For each page allocated in a task, a count is maintained of the number of times that the task has wired the page. The expression $\underline{w}ire\_count(task, page\_index)$ denotes the number of times that $task$ has wired the page indicated by $page\_index$ in its address space. As long as a task's count for $page\_index$ remains nonzero, the physical page associated with $page\_index$ must be retained in memory. In other words, a physical page may only be evicted when no task has the page wired. The set $wired$ denotes the set of physical pages that are wired by some task.

Mach Definition 61

Wired

$$AddressSpace$$
$$PageAndMemory$$
$$\underline{w}ire\_count : (TASK \times PAGE\_INDEX) \rightarrowtail\!\!\!\!\!+ \mathbb{N}$$
$$wired\_locations : \mathbb{P}(TASK \times PAGE\_INDEX)$$
$$wired : \mathbb{P}\, PAGE$$

$$\mathrm{dom}\, \underline{w}ire\_count = \underline{a}llocated$$
$$wired\_locations = \{ task : TASK;\ page\_index : PAGE\_INDEX$$
$$\quad \mid \underline{w}ire\_count(task, page\_index) > 0 \}$$
$$wired\_locations \subseteq \mathrm{dom}(representing\_page \circ \underline{m}ap\_rel)$$
$$wired = (representing\_page \circ \underline{m}ap\_rel)\, (\!| wired\_locations |\!)$$

Review Note:
The $\underline{w}ire\_count$ component corresponds to the VM entry wire count. A page is wired if any VM entry that is mapped to it is wired. For efficiency the prototype maintains two wire counts, one on VM entries and another on pages. The latter denotes the number of VM entries that have the page wired ignoring multiple wirings by a single VM entry. We do not model the page wire count.

4.9.8 Summary

The memory system is comprised of memory objects, address spaces, pages, and backing chains.

Mach Definition 62

$MemorySystem$

$Memory$
$AddressSpace$
$PageAndMemory$
$MachProtection$
$Lock$
$ShadowMemories$
$Inheritance$
$Wired$

$\underline{a}llocated = \mathrm{dom}\ \underline{m}ach\_protection$
$\mathrm{ran}\ represented\_memory \subseteq \mathrm{dom}\ object\_port$
$\forall\ task\_va\_pair : TASK \times PAGE\_INDEX$
$\mid task\_va\_pair \in \mathrm{dom}\ \underline{m}ap\_rel$
$\wedge\ \underline{m}ap\_rel(task\_va\_pair) \in \mathrm{dom}\ representing\_page$
$\bullet\ \underline{m}ach\_protection(task\_va\_pair)$
$\subseteq PROTECTION \setminus page\_locks(representing\_page(\underline{m}ap\_rel(task\_va\_pair)))$
$\mathrm{dom}\ \underline{i}nheritance = \underline{a}llocated$
$mapped \subseteq \mathrm{dom}\ object\_port$

4.10 Messages

This section discusses the structure of messages.

4.10.1 Message Options

The type $MACH\_MSG\_OPTION$ denotes the base values of the $options$ parameter of **mach_msg**. The recognized values of this type are $Mach\_send\_msg$, $Mach\_rcv\_msg$, $Mach\_send\_cancel$, $Mach\_send\_notify$, $Mach\_rcv\_notify$, $Mach\_rcv\_large$, $Mach\_send\_timeout$, and $Mach\_rcv\_timeout$. The $options$ parameter is set to some set of the base values.

Mach Definition 63

$[MACH\_MSG\_OPTION]$

$$\begin{array}{|l}
Mach_send_msg : MACH_MSG_OPTION \\
Mach_rcv_msg : MACH_MSG_OPTION \\
Mach_send_cancel : MACH_MSG_OPTION \\
Mach_send_notify : MACH_MSG_OPTION \\
Mach_rcv_notify : MACH_MSG_OPTION \\
Mach_rcv_large : MACH_MSG_OPTION \\
Mach_send_timeout : MACH_MSG_OPTION \\
Mach_rcv_timeout : MACH_MSG_OPTION \\
\hline
\mathsf{disjoint}\ \langle \{Mach_send_msg\}, \{Mach_rcv_msg\}, \\
\quad \{Mach_send_cancel\}, \{Mach_rcv_large\}, \{Mach_send_notify\}, \\
\quad \{Mach_rcv_notify\}, \{Mach_send_timeout\}, \{Mach_rcv_timeout\}\rangle
\end{array}$$

4.10.2 Complex Messages

In addition to simply carrying data, a message can also carry port rights and memory regions. A message carrying port rights or memory regions is called a *complex* message. Each message carries a flag indicating whether the message contains port rights or memory regions. The type $COMPLEX\_OPTION$ consists of the elements $Co\_carries\_rights$ and $Co\_carries\_memory$; the flag carried in each message is a set of these values. Note that a flag containing both elements indicates that the message contains both port rights and memory regions.

Mach Definition 64

$$[COMPLEX\_OPTION]$$

$$\begin{array}{|l}
Co_carries_rights : COMPLEX_OPTION \\
Co_carries_memory : COMPLEX_OPTION \\
\hline
\mathsf{disjoint}\ \langle \{Co_carries_rights\}, \{Co_carries_memory\}\rangle
\end{array}$$

4.10.3 Data Types

Each element in the body of a message is typed. The set $MACH\_MSG\_TYPE$ denotes the set of data types recognized by the system.

Mach Definition 65

$$[MACH\_MSG\_TYPE]$$

Whenever a port right is sent in a message, the client indicates a transfer option for the port right. The collection of acceptable transfer options is denoted by $Recognized\_transfer\_options$ and contain the values Mmt_make_send, Mmt_copy_send, Mmt_move_send, $Mmt\_make\_send\_once$, $Mmt\_move\_send\_once$, and $Mmt\_move\_receive$.

An element of type Mmt_make_send indicates a receive right held by the sender from which a send right is to be created for the receiver. Similarly, an element of type $Mmt\_make\_send\_once$ indicates a receive right held by the sender from which a send-once right is to be created for the receiver.

An element of type Mmt_copy_send indicates a send right that should be copied from the sender's port name space into the receiver's port name space. In other words, the sender retains the existing port right while passing the right to the receiver.

An element of type Mmt_move_send indicates a send right that should be moved from the sender's port name space into the receiver's port name space. In other words, the sender's reference count is decremented by one and the receiver's reference count is incremented by one. If the sender's reference count was one, then the sender loses the capability associated with the right. If the receiver's reference count was zero, then the receiver gains the capability associated with the right. Similarly, $Mmt\_move\_send\_once$ and $Mmt\_move\_receive$ allow send-once and receive rights to be moved from the sender's name space to the receiver's name space.

Mach Definition 66

$$
\begin{array}{l}
Mmt\_make\_send : MACH\_MSG\_TYPE \\
Mmt\_copy\_send : MACH\_MSG\_TYPE \\
Mmt\_move\_send : MACH\_MSG\_TYPE \\
Mmt\_make\_send\_once : MACH\_MSG\_TYPE \\
Mmt\_move\_send\_once : MACH\_MSG\_TYPE \\
Mmt\_move\_receive : MACH\_MSG\_TYPE \\
Recognized\_transfer\_options : \mathbb{P}\ MACH\_MSG\_TYPE \\
\hline
\langle \{Mmt\_make\_send\}, \{Mmt\_copy\_send\}, \{Mmt\_move\_send\}, \\
\quad \{Mmt\_make\_send\_once\}, \{Mmt\_move\_send\_once\}, \{Mmt\_move\_receive\}\rangle \\
\qquad \mathsf{partition}\ Recognized\_transfer\_options
\end{array}
$$

After the kernel translates the port rights to an internal representation, it is no longer relevant whether the right was moved, copied or made and the kernel simply records the type of right, $Mach\_msg\_type\_port\_receive$, $Mach\_msg\_type\_port\_send$, or $Mach\_msg\_type\_port\_send\_once$. These values of $MACH\_MSG\_TYPE$ comprise the set $Mach\_msg\_type\_port\_rights$.

Mach Definition 67

$$
\begin{array}{l}
Mach\_msg\_type\_port\_receive : MACH\_MSG\_TYPE \\
Mach\_msg\_type\_port\_send : MACH\_MSG\_TYPE \\
Mach\_msg\_type\_port\_send\_once : MACH\_MSG\_TYPE \\
Mach\_msg\_type\_port\_rights : \mathbb{P}\ MACH\_MSG\_TYPE \\
\hline
\langle \{Mach\_msg\_type\_port\_receive\}, \{Mach\_msg\_type\_port\_send\}, \\
\quad \{Mach\_msg\_type\_port\_send\_once\}\rangle \\
\mathsf{partition}\ Mach\_msg\_type\_port\_rights
\end{array}
$$

4.10.4 Message Headers

The header for a message residing in user-space memory or kernel-space memory contains the following data:

- $local\_port$ — specifies the reply port when sending a message ($Mach\_port\_null$ indicates no reply port is specified)
- $local\_rights$ — the port rights for the local port (if one is specified)
- $remote\_port$ — specifies the destination port when sending a message
- $remote\_rights$ — the port rights for the remote port
- $size$ — specifies the size, in bytes, of a message when receiving
- $msg\_sequence\_no$ — specifies the sequence number when receiving a message
- $operation$ — operation or function id set by message sender

In addition, a message header in kernel space contains a value $complex$ which indicates whether the message carries port rights or memory regions or both. This value is a set of elements of type $COMPLEX\_OPTION$. In place of $complex$, a message header in user space contains a single value $complex\_boolean$ indicating whether the message carries port rights and/or memory regions. The possible values are $Co\_carries\_rights\_and\_or\_memory$ and $Co\_carries\_neither\_rights\_nor\_memory$. If $complex\_boolean$ has value $Co\_carries\_neither\_rights\_nor\_memory$, then the message contains no port rights nor memory regions regardless of what is indicated by the individual data elements of the message.

Mach Definition 68

$[OPERATION]$

$COMPLEX\_OPTION\_BOOLEAN$
$::= Co\_carries\_rights\_and\_or\_memory$
$\mid Co\_carries\_neither\_rights\_nor\_memory$

$MachMsgHeader$
$local\_port : NAME$
$local\_rights : \mathbb{P}\ MACH\_MSG\_TYPE$
$remote\_port : NAME$
$remote\_rights : MACH\_MSG\_TYPE$
$size : \mathbb{N}$
$msg\_sequence\_no : \mathbb{N}$
$operation : OPERATION$
$complex\_boolean : COMPLEX\_OPTION\_BOOLEAN$

$\#local\_rights \leq 1$

Messages residing in kernel space contain ports rather than names. Thus, the$remote_port$ and $local\_port$ fields contain ports instead of names when a message is in transit. If$Mach_port_null$ was specified as the name of the local port in the $MachMsgHeader$, then $local\_port$ is empty in the corresponding $MachInternalHeader$.

Mach Definition 69

$MachInternalHeader$
$local\_port : \mathbb{P}\ PORT$
$local\_rights : \mathbb{P}\ MACH\_MSG\_TYPE$
$remote\_port : PORT$
$remote\_rights : MACH\_MSG\_TYPE$
$size : \mathbb{N}$
$msg\_sequence\_no : \mathbb{N}$
$operation : OPERATION$
$complex : \mathbb{P}\ COMPLEX\_OPTION$

$\#local\_rights = \#local\_port \leq 1$

4.10.5 Outcall Operations

There are several sets of operation identifiers used in messages to external servers (e.g., the security server) and user tasks. Some of these identifiers are used by the kernel when sending outcalls. We use

- $Exception\_ids$ to denote the set of operations used by the kernel when sending an exception message, The only element of this set is $Mach\_exception\_id$.
- $Kernel\_service\_reply\_ids$ to denote the set of operations used by the kernel in reply messages to kernel service requests,
- $Security\_server\_ids$ to denote the set of security server operations,
- $Audit\_ids$ to denote the set of audit operations,
- $Mem\_obj\_confirmation\_ids$ to denote the set of operations used by the kernel when sending confirmations of memory operations to a pager,
- $Pager\_request\_ids$ to denote the set of pager operations,
- $Mach\_notify\_ids$ to denote the set of operations used by the kernel in notification messages, and
- $Network\_packet\_ids$ to denote the set of operations used by the kernel when forwarding network packets.

We give a partial description of the identifiers in these sets.

Mach Definition 70

$$\begin{array}{l} Exception\_ids : \mathbb{P}\ OPERATION \\ Mach\_exception\_id : OPERATION \\ \hline Exception\_ids = \{Mach\_exception\_id\} \end{array}$$

Mach Definition 71

$$Kernel\_service\_reply\_ids : \mathbb{P}\ OPERATION$$

Mach Definition 72

$$\begin{array}{l} Security\_server\_ids : \mathbb{P}\ OPERATION \\ SSI\_compute\_av\_id : \\ \quad OPERATION \\ \hline \{SSI\_compute\_av\_id\} \\ \quad \subseteq Security\_server\_ids \end{array}$$

Mach Definition 73

$$\begin{array}{l} Audit\_ids : \mathbb{P}\ OPERATION \\ Audit\_batch\_id, Audit\_id : \\ \quad OPERATION \\ \hline \{Audit\_batch\_id, Audit\_id\} \\ \quad \subseteq Audit\_ids \end{array}$$

Mach Definition 74

$$\begin{array}{l} Mem\_obj\_confirmation\_ids : \mathbb{P}\ OPERATION \\ Memory\_object\_change\_completed\_id, Memory\_object\_lock\_completed\_id, \\ Memory\_object\_supply\_completed\_id : \\ \quad OPERATION \\ \hline \{Memory\_object\_change\_completed\_id, Memory\_object\_lock\_completed\_id, \\ \qquad Memory\_object\_supply\_completed\_id\} \\ \quad \subseteq Mem\_obj\_confirmation\_ids \end{array}$$

Mach Definition 75

$$
\begin{array}{l}
Pager\_request\_ids : \mathbb{P}\ OPERATION \\
Memory\_object\_copy\_id, Memory\_object\_create\_id, Memory\_object\_data\_initialize\_id, \\
Memory\_object\_data\_request\_id, Memory\_object\_data\_return\_id, \\
Memory\_object\_data\_unlock\_id, Memory\_object\_data\_write\_id, \\
Memory\_object\_init\_id, Memory\_object\_terminate\_id : \\
\quad OPERATION \\
\hline
\{Memory\_object\_copy\_id, Memory\_object\_create\_id, Memory\_object\_data\_initialize\_id, \\
\qquad Memory\_object\_data\_request\_id, Memory\_object\_data\_return\_id, \\
\qquad Memory\_object\_data\_unlock\_id, Memory\_object\_data\_write\_id, \\
\qquad Memory\_object\_init\_id, Memory\_object\_terminate\_id\} \\
\quad \subseteq Pager\_request\_ids
\end{array}
$$

Mach Definition 76

$$
\begin{array}{l}
Mach\_notify\_ids : \mathbb{P}\ OPERATION \\
Ipc\_notify\_dead\_name\_id, Ipc\_notify\_msg\_accepted\_id, Ipc\_notify\_no\_senders\_id, \\
Ipc\_notify\_port\_deleted\_id, Ipc\_notify\_port\_destroyed\_id, \\
Ipc\_notify\_send\_once\_id : \\
\quad OPERATION \\
\hline
\{Ipc\_notify\_dead\_name\_id, Ipc\_notify\_msg\_accepted\_id, Ipc\_notify\_no\_senders\_id, \\
\qquad Ipc\_notify\_port\_deleted\_id, Ipc\_notify\_port\_destroyed\_id, \\
\qquad Ipc\_notify\_send\_once\_id\} \\
\quad \subseteq Mach\_notify\_ids
\end{array}
$$

Mach Definition 77

$$
\begin{array}{l}
Network\_packet\_ids : \mathbb{P}\ OPERATION \\
Forward\_net\_packet\_id : \\
\quad OPERATION \\
\hline
\{Forward\_net\_packet\_id\} \\
\quad \subseteq Network\_packet\_ids
\end{array}
$$

4.10.6 Message Bodies

The body of a message consists of a sequence of message elements. Each element contains the following:

- the number of data elements contained in the message element
- a data type
- a collection of data elements or a single address

A triple that contains a collection of data elements represents in-line data. The number of data elements in the collection is the same as the specified number of data elements, and each such element is of the specified type. A triple that contains a single address represents out-of-line data. The address specifies the start of the area of memory containing the data. The data in that area is interpreted as being a collection of the specified number of data elements of the specified data type. Each out-of-line element contains a flag indicating whether the memory should be deallocated from the sender's address space. The possible values of this flag are $Msg\_deallocate$ and $Msg\_dont\_deallocate$.

Mach Definition 78

$$[MSG\_DATA]$$

$$OLSD ::= Msg\_deallocate \mid Msg\_dont\_deallocate$$
$$BASE\_MSG\_ELEMENT$$
$$::= In\_line \langle\langle \mathbb{N} \times MACH\_MSG\_TYPE \times \text{seq}\, MSG\_DATA \rangle\rangle$$
$$\mid Out\_of\_line \langle\langle \mathbb{N} \times MACH\_MSG\_TYPE \times VIRTUAL\_ADDRESS \times OLSD \rangle\rangle$$

Thus, an in-line message element is denoted by:

$$In\_line(n, mach\_msg\_type, data\_seq)$$

and an out-of-line message element is denoted by:

$$Out\_of\_line(n, mach\_msg\_type, va, olsd)$$

The number of entries specified in a triple representing in-line data must be the same as the number of entries in the specified sequence of data elements. The set $Msg\_element$ denotes the set of valid message elements, and the set $MESSAGE\_BODY$ denotes the set of sequences of valid message elements. In other words, $MESSAGE\_BODY$ denotes the set of valid message bodies.

Mach Definition 79

$$Msg\_element : \mathbb{P}\, BASE\_MSG\_ELEMENT$$

$$\begin{aligned} & Msg\_element \\ & = \{ msg\_element : BASE\_MSG\_ELEMENT \\ & \quad \mid (\exists\, n : \mathbb{N};\ mach\_msg\_type : MACH\_MSG\_TYPE;\ data\_seq : \text{seq}\, MSG\_DATA; \\ & \quad va : VIRTUAL\_ADDRESS;\ olsd : OLSD \\ & \quad \bullet (msg\_element = In\_line(n, mach\_msg\_type, data\_seq) \\ & \qquad \wedge \#data\_seq = n) \\ & \quad \vee\ msg\_element = Out\_of\_line(n, mach\_msg\_type, va, olsd)) \} \end{aligned}$$

Mach Definition 80

$$MESSAGE\_BODY == \text{seq}\, Msg\_element$$

When a message is moved into kernel space, the port names appearing in the message are transformed into port identifiers and the virtual addresses indicating out-of-line data are transformed into memory-offset pairs. In other words, the client specific names for kernel entities are transformed into the appropriate global names used internal to the kernel. Thus, an element in a message body in kernel space is of one of the following forms:

- $Msg\_value(n, mach\_msg\_type, (task, value\_seq))$ — an in-line element; if $mach\_msg\_type$ is an element of $Recognized\_transfer\_options$ and some elements of $value\_seq$ have not yet been resolved to ports then further processing is required to transform the sequence of data into a sequence of ports.
  Note that there are two forms for elements of $value\_seq$. An entry of the form $V\_data(msg\_data, v\_data\_l)$ denotes the data msg_data while an entry of the form

$V\_port(port, v\_data\_l)$ denotes a port name that has been resolved into a port. In either case, v_data_l indicates whether the element came from an in-line data element or an out-of-line data element. The only time v_data_l will indicate an out-of-line data element is when the element is a port name from an out-of-line data element that has been resolved into a port.

- $Transit\_right(n, mach\_msg\_type, (task, port\_seq, v\_data\_l))$ — a sequence of port rights in transit; $task$ indicates the task that sent the message and v_data_l indicates whether the port right was sent in-line or out-of-line
- $Msg\_region(n, mach\_msg\_type, (task, va, olsd))$ — an out-of-line element that requires further processing to transform the task-address pair into a memory-offset pair; $task$ indicates the task that sent the message and $olsd$ indicates whether the region should be deallocated from $task$'s address space
- $Transit\_memory(n, mach\_msg\_type, (task, memory, offset))$ — an out-of-line element that has been transformed from a task-address pair to a memory-offset pair; $task$ indicates the task that sent the message

The number of entries specified in a triple representing in-line data must be the same as the number of entries in the specified sequence of data elements. The type $Internal\_element$ denotes the set of valid message elements internal to the kernel, and the type $INTERNAL\_BODY$ denotes the set of sequences of these elements. Thus, $INTERNAL\_BODY$ denotes the set of message bodies that can be stored in the kernel.

Mach Definition 81

$$
\begin{aligned}
&V\_DATA\_LOCATION ::= V\_data\_in \mid V\_data\_out\\
&MSG\_VALUE ::= V\_data\langle\langle MSG\_DATA \times V\_DATA\_LOCATION\rangle\rangle\\
&\quad \mid V\_port\langle\langle PORT \times V\_DATA\_LOCATION\rangle\rangle\\
&BASE\_INTERNAL\_ELEMENT\\
&\quad ::= Msg\_value\langle\langle \mathbb{N} \times MACH\_MSG\_TYPE \times (TASK \times \operatorname{seq} MSG\_VALUE)\rangle\rangle\\
&\quad \mid Transit\_right\langle\langle \mathbb{N} \times MACH\_MSG\_TYPE\\
&\qquad \times (TASK \times \operatorname{seq} PORT \times V\_DATA\_LOCATION)\rangle\rangle\\
&\quad \mid Msg\_region\langle\langle \mathbb{N} \times MACH\_MSG\_TYPE \times (TASK \times VIRTUAL\_ADDRESS \times OLSD)\rangle\rangle\\
&\quad \mid Transit\_memory\langle\langle \mathbb{N} \times MACH\_MSG\_TYPE \times (TASK \times MEMORY \times OFFSET)\rangle\rangle
\end{aligned}
$$

Editorial Note:
$Transit\_right$ probably needs to be considered in the following.

$$
\begin{aligned}
&Internal\_element : \mathbb{P}\, BASE\_INTERNAL\_ELEMENT\\
\hline
&Internal\_element\\
&= \{msg\_element : BASE\_INTERNAL\_ELEMENT\\
&\quad \mid (\exists\, n : \mathbb{N};\ mach\_msg\_type : MACH\_MSG\_TYPE;\ task : TASK;\\
&\qquad value\_seq : \operatorname{seq} MSG\_VALUE;\ port\_seq : \operatorname{seq} PORT;\\
&\qquad memory : MEMORY;\ offset : OFFSET;\ va : VIRTUAL\_ADDRESS;\\
&\qquad olsd : OLSD;\ v\_data\_l : V\_DATA\_LOCATION\\
&\qquad \bullet (msg\_element = Msg\_value(n, mach\_msg\_type, (task, value\_seq))\\
&\qquad\quad \wedge \#value\_seq = n)\\
&\qquad \vee msg\_element = Msg\_region(n, mach\_msg\_type, (task, va, olsd))\\
&\qquad \vee msg\_element\\
&\qquad\quad = Transit\_memory(n, mach\_msg\_type, (task, memory, offset)))\}
\end{aligned}
$$

$$
\begin{array}{l}
INTERNAL\_BODY == \{ body : \operatorname{seq} Internal\_element \\
\quad \mid (\exists\, task : TASK \\
\qquad \bullet (\forall\, n : \mathbb{N};\ mach\_msg\_type : MACH\_MSG\_TYPE; \\
\qquad\quad value\_seq : \operatorname{seq} MSG\_VALUE; \\
\qquad\quad olsd : OLSD;\ task_1 : TASK;\ va : VIRTUAL\_ADDRESS \\
\qquad\quad \mid Msg\_value(n, mach\_msg\_type, (task_1, value\_seq)) \in \operatorname{ran} body \\
\qquad\quad \lor Msg\_region(n, mach\_msg\_type, (task_1, va, olsd)) \in \operatorname{ran} body \\
\qquad\quad \bullet task = task_1))\}
\end{array}
$$

Review Note:
Should $Transit\_memory$ be added to the above?

Note that all of the elements in a single message body must contain the same task identifier. It is intended that this task identifier unambiguously defines the identity of the task that sent the message.

4.10.7 Message Status

Once a message enters the kernel, it can be in one of three states:

- Msg_stat_send — indicates that the kernel is performing processing to send the message
- Msg_stat_pseudo — indicates that the kernel is performing processing to return the message to the message sender as part of a failed send request
- Msg_stat_rcv — indicates that the kernel is performing processing to receive the message

These elements comprise the values of the type MSG_STATUS.

The following error conditions can arise during the processing of a message: $Msg\_error\_invalid\_memory$, $Msg\_error\_invalid\_right$, $Msg\_error\_invalid\_type$, $Msg\_error\_msg\_too\_small$, $Msg\_error\_notify\_in\_progress$, and $Msg\_error\_timed\_out$. These values comprise the set MSG_ERROR.

Mach Definition 82

$$MSG\_STATUS ::= Msg\_stat\_send \mid Msg\_stat\_pseudo \mid Msg\_stat\_rcv$$

$$
\begin{array}{l}
MSG\_ERROR ::= Msg\_error\_invalid\_memory \mid Msg\_error\_invalid\_right \\
\quad \mid Msg\_error\_invalid\_type \mid Msg\_error\_msg\_too\_small \\
\quad \mid Msg\_error\_notify\_in\_progress \mid Msg\_error\_timed\_out
\end{array}
$$

4.10.8 Message Structure

Each message is modeled as containing fields $header$ and $body$. The type $Message$ denotes the set of user space messages.

Mach Definition 83

$$
\begin{array}{|l}
\hline
Message \\
header : MachMsgHeader \\
body : MESSAGE\_BODY \\
\hline
\end{array}
$$

In addition to the header and body, messages in transit also contain the following fields:

- $option$ — indicates the options specified by the client
- $time\_out\_at$ — indicates when a given send or receive request will time out
  If the set contained in this field is empty, then the message will not time out. Otherwise, the set contains exactly one value and this value defines the earliest time at which the associated send or receive request can time out.
- $status$ — indicates future processing the kernel must perform on the message
- $error$ — indicates the first error (if any) that occurred during the processing of the message.

Editorial Note:
$status$ and $error$ should be removed as the purpose they were intended to serve can now be accomplished more generally using the tools of the execution model.

The type $InternalMessage$ denotes the possible values of messages in transit.

Mach Definition 84

$$
\begin{array}{l}
\textbf{schema } InternalMessage \\
\quad header : MachInternalHeader \\
\quad body : INTERNAL\_BODY \\
\quad option : \mathbb{P}\ MACH\_MSG\_OPTION \\
\quad time\_out\_at : \mathbb{P}\ \mathbb{N} \\
\quad status : MSG\_STATUS \\
\quad error : \mathbb{P}\ MSG\_ERROR \\
\hline
\quad \#time\_out\_at \leq 1 \\
\quad \#error \leq 1
\end{array}
$$

4.10.9 Pending Receives

Each port can have clients blocked on message receive requests waiting for messages to arrive at the port. Each pending receive request has the following associated information:

- $notify$ — the notify port name specified by the receiving task
- $option$ — the options specified by the receiving task
- rcv_size — the receive size specified by the receiving task
- $time\_out\_at$ — the time at which the request will time out; this has the same format as the $time\_out\_at$ component of $InternalMessage$.

Mach Definition 85

$$
\begin{array}{l}
\textbf{schema } PendingReceive \\
\quad notify : NAME \\
\quad option : \mathbb{P}\ MACH\_MSG\_OPTION \\
\quad rcv\_size : \mathbb{N} \\
\quad time\_out\_at : \mathbb{P}\ \mathbb{N} \\
\hline
\quad \#time\_out\_at \leq 1
\end{array}
$$

4.10.10 Reply Ports

The sender of a message can specify a reply port for the receiver to use to reply to the message. The sender does so by setting the $local\_port$ field to its name for the port. For convenience, the relation $\underline{r}eply\_port\_rel$ is used to denote the reply port and transferred right in a message specifying a reply port. The interpretation of:

$$(message, (port, right))$$

being an element of $\underline{r}eply\_port\_rel$ is that $message$ transfers the type of right specified by $right$ (send or send-once) for $port$ to the receiver of $message$. The intent is that the receiver use the transferred right to send a reply message to $port$. Each message contains at most one reply port and right for that port. For convenience, the expressions $reply\_port(message)$ and $reply\_port\_right(message)$ are used to denote the reply port and transferred right contained in a given message.

Mach Definition 86

$$
\begin{array}{l}
\textit{ReplyPorts} \\
\hline
\underline{r}eply\_port\_rel : MESSAGE \leftrightarrow (PORT \times \{Send, Send\_once\}) \\
reply\_port : MESSAGE \nrightarrow PORT \\
reply\_port\_right : MESSAGE \nrightarrow \{Send, Send\_once\} \\
\hline
reply\_port = \{ message : MESSAGE;\ port : PORT;\ right : RIGHT \\
\quad \mid (message, (port, right)) \in \underline{r}eply\_port\_rel \bullet (message, port)\} \\
reply\_port\_right = \{ message : MESSAGE;\ port : PORT;\ right : RIGHT \\
\quad \mid (message, (port, right)) \in \underline{r}eply\_port\_rel \bullet (message, right)\}
\end{array}
$$

4.10.11 Summary

This section has defined the data structures used to model messages. The expression $\underline{m}sg\_contents(message)$ is used to denote the internal message structure associated with each message identifier, and the expression $\underline{p}ending\_receives(task, name)$ indicates the receive requests currently pending for threads in $task$ that attempted to receive through the port named by $name$. The expression $\underline{t}ask\_received\_msgs(task)$ denotes the set of user-space messages that have been received by $task$.

For convenience, the expression $msg\_operation(message)$ is used to denote the type of operation requested by $message$. In other words, the returned value is the $operation$ field of the message identified by $message$.

Mach Definition 87

$$
\begin{array}{l}
\textit{Operations} \\
\hline
msg\_operation : MESSAGE \nrightarrow OPERATION
\end{array}
$$

Mach Definition 88

$$
\begin{array}{l}
\textbf{Messages} \\
\hline
TaskExist \\
MessageExist \\
Operations \\
ReplyPorts \\
\underline{m}sg\_contents : MESSAGE \nrightarrow InternalMessage \\
\underline{p}ending\_receives : TASK \times NAME \nrightarrow \operatorname{seq} PendingReceive \\
\underline{t}ask\_received\_msgs : TASK \longrightarrow \mathbb{P}\, MESSAGE \\
\hline
\operatorname{dom} reply\_port \subseteq \underline{m}essage\_exists \\
\operatorname{dom} msg\_operation = \operatorname{dom} \underline{m}sg\_contents = \underline{m}essage\_exists \\
\forall\, message : MESSAGE \mid message \in \underline{m}essage\_exists \\
\bullet\, msg\_operation(message) = (\underline{m}sg\_contents(message)).header.operation \\
\forall\, message : MESSAGE;\ port : PORT \mid message \in \underline{m}essage\_exists \\
\bullet\, (message, (port, Send)) \in \underline{r}eply\_port\_rel \\
\Leftrightarrow ((\underline{m}sg\_contents(message)).header.local\_port = \{port\} \\
\quad \wedge\, (\underline{m}sg\_contents(message)).header.local\_rights \\
\qquad \cap \{\, Mmt\_make\_send, Mmt\_move\_send, Mmt\_copy\_send \,\} \neq \emptyset) \\
\forall\, message : MESSAGE;\ port : PORT \mid message \in \underline{m}essage\_exists \\
\bullet\, (message, (port, Send\_once)) \in \underline{r}eply\_port\_rel \\
\Leftrightarrow ((\underline{m}sg\_contents(message)).header.local\_port = \{port\} \\
\qquad \wedge\, (\underline{m}sg\_contents(message)).header.local\_rights \\
\qquad\quad \cap \{\, Mmt\_make\_send\_once, Mmt\_move\_send\_once \,\} \neq \emptyset) \\
\forall\, task : TASK \\
\mid task \notin \underline{t}ask\_exists \\
\bullet\, \underline{t}ask\_received\_msgs(task) = \emptyset
\end{array}
$$

Review Note:
Must figure out what the axioms are on $\underline{p}ending\_receives$.

4.11 Processors and Processor Sets

Each host has a default processor set denoted by $\underline{d}efault$. Furthermore, each host has a master processor denoted by $\underline{m}aster\_proc$.

Mach Definition 89

$$
\begin{array}{l}
\textbf{HostsAndProcessors} \\
\hline
ProcessorsAndPorts \\
\underline{d}efault : PROCESSOR\_SET \\
\underline{m}aster\_proc : PROCESSOR \\
\hline
\underline{d}efault \in \operatorname{dom} \underline{p}s\_control\_port\_rel \\
\underline{m}aster\_proc \in \operatorname{dom} \underline{p}rocessor\_port\_rel
\end{array}
$$

Each processor is a member of a single processor set. The relation $\underline{m}ember\_rel$ indicates which processors belong to each processor set. For convenience, the expressions $processors(procset)$ and $proc\_assigned\_procset(proc)$ are used to denote, respectively, the set of processors that belong to $procset$ and the processor set to which $proc$ belongs.

Mach Definition 90

$$
\begin{array}{l}
\textit{ProcessorAndProcessorSet} \\
\hline
\textit{ProcessorsAndPorts} \\
\underline{m}ember\_rel : PROCESSOR \leftrightarrow PROCESSOR\_SET \\
processors : PROCESSOR\_SET \longrightarrow \mathbb{P}\, PROCESSOR \\
proc\_assigned\_procset : PROCESSOR \nrightarrow PROCESSOR\_SET \\
\hline
\mathrm{dom}\, \underline{m}ember\_rel \subseteq \mathrm{dom}\, \underline{p}rocessor\_port\_rel \\
\mathrm{ran}\, \underline{m}ember\_rel \subseteq \mathrm{dom}\, \underline{p}s\_control\_port\_rel \\
proc\_assigned\_procset = \underline{m}ember\_rel \\
processors = (\lambda\, procset : PROCESSOR\_SET \bullet \underline{m}ember\_rel^{\sim} (\!| \{procset\} |\!))
\end{array}
$$

Each task is assigned to a single processor set. The relation $\underline{t}ask\_assignment\_rel$ indicates the association between tasks and processor sets. For convenience, the expressions $have\_assigned\_tasks(procset)$ and $task\_assigned\_to(task)$ are used to denote, respectively, the set of tasks assigned to $procset$ and processor set to which $task$ is assigned.

Mach Definition 91

$$
\begin{array}{l}
\textit{TaskAndProcessorSet} \\
\hline
\textit{SpecialTaskPorts} \\
\textit{ProcessorsAndPorts} \\
\underline{t}ask\_assignment\_rel : TASK \leftrightarrow PROCESSOR\_SET \\
have\_assigned\_tasks : PROCESSOR\_SET \longrightarrow \mathbb{P}\, TASK \\
task\_assigned\_to : TASK \nrightarrow PROCESSOR\_SET \\
\hline
\mathrm{dom}\, \underline{t}ask\_assignment\_rel = \mathrm{ran}\, self\_task \\
\mathrm{ran}\, \underline{t}ask\_assignment\_rel \subseteq \mathrm{dom}\, \underline{p}s\_control\_port\_rel \\
\underline{t}ask\_assignment\_rel = task\_assigned\_to \\
have\_assigned\_tasks = (\lambda\, procset : PROCESSOR\_SET \\
\quad \bullet \underline{t}ask\_assignment\_rel^{\sim} (\!| \{procset\} |\!))
\end{array}
$$

Similarly, Each thread is assigned to a single processor set. The relation $\underline{t}hread\_assignment\_rel$ associates threads with processor sets. For convenience, the expressions $have\_assigned\_threads(procset)$ and $thread\_assigned\_to(thread)$ are used to denote, respectively, the set of threads assigned to $procset$ and processor set to which $thread$ is assigned.

Each processor set has a set of enabled scheduling policies, denoted by $\underline{e}nabled\_sp(procset)$ and a maximum priority for assigned threads, denoted by $\underline{p}s\_max\_priority(procset)$. The set of enabled scheduling policies for a thread's processor set is used to constrain the policies that can be assigned to that thread. The maximum scheduling priority for a processor set constrains the priorities that can be assigned to a newly created thread associated with that processor set.

Mach Definition 92

$$
\begin{array}{l}
\textbf{ThreadAndProcessorSet} \\
\hline
ProcessorSetExist \\
ProcessorsAndPorts \\
SpecialThreadPorts \\
ThreadSchedPolicy \\
\underline{t}hread\_assignment\_rel : THREAD \leftrightarrow PROCESSOR\_SET \\
have\_assigned\_threads : PROCESSOR\_SET \longrightarrow \mathbb{P}\ THREAD \\
thread\_assigned\_to : THREAD \rightarrowtail\!\!\!\!+ PROCESSOR\_SET \\
\underline{e}nabled\_sp : PROCESSOR\_SET \rightarrowtail\!\!\!\!+ \mathbb{P}\ SCHED\_POLICY \\
\underline{p}s\_max\_priority : PROCESSOR\_SET \rightarrowtail\!\!\!\!+ \mathbb{Z} \\
\hline
\underline{t}hread\_assignment\_rel = thread\_assigned\_to \\
have\_assigned\_threads = (\lambda\, procset : PROCESSOR\_SET \\
\quad \bullet\ \underline{t}hread\_assignment\_rel^{\sim} (\!| \{procset\} |\!)) \\
\mathrm{dom}\ \underline{t}hread\_assignment\_rel = \mathrm{dom}\ thread\_self \\
\mathrm{ran}\ \underline{t}hread\_assignment\_rel \subseteq \mathrm{dom}\ \underline{p}s\_control\_port\_rel \\
\mathrm{dom}\ \underline{e}nabled\_sp = \mathrm{dom}\ \underline{p}s\_max\_priority = \underline{p}rocset\_exists \\
\bigcup(\mathrm{ran}\ \underline{e}nabled\_sp) \subseteq \underline{s}upported\_sp \\
\mathrm{ran}\ \underline{p}s\_max\_priority \subseteq Priority\_levels
\end{array}
$$

Each processor may have an active thread. The expression $\underline{a}ctive\_thread(proc)$ indicates the thread (if any) that is active on $proc$.

Mach Definition 93

$$
\begin{array}{l}
\textbf{ThreadsAndProcessors} \\
\hline
ThreadExist \\
Exist \\
\underline{a}ctive\_thread : PROCESSOR \rightarrowtail\!\!\!\!+ THREAD \\
\hline
\mathrm{dom}\ \underline{a}ctive\_thread \subseteq \underline{p}roc\_exists \\
\mathrm{ran}\ \underline{a}ctive\_thread \subseteq \underline{t}hread\_exists
\end{array}
$$

4.12 Time

Each host provides a system clock. The current system time is denoted by $\underline{h}ost\_time$.

Mach Definition 94

$$
\begin{array}{l}
\textbf{HostTime} \\
\hline
\underline{h}ost\_time : \mathbb{N}
\end{array}
$$

4.13 Devices

Each device has an associated count indicating how many times the device has been opened and not closed. We use $\underline{d}evice\_open\_count(dev)$ to indicate the count associated with dev. This count is incremented each time dev is opened and decremented each time dev is closed. Each device with a positive creation count has an associated device port that represents the device.

Mach Definition 95

$$\begin{array}{l} \textit{DeviceOpenCount} \\ \hline DevicesAndPorts \\ \underline{d}evice\_open\_count : DEVICE \longrightarrow \mathbb{N} \\ \hline \mathrm{dom}\ device\_port = \{\ dev : DEVICE \mid \underline{d}evice\_open\_count(dev) > 0\ \} \\ \hline \end{array}$$

A kernel-space device driver may supply event counters for use by user-space device drivers. An event counter is used as a semaphore for events produced by kernel-space drivers. The counter is incremented when a relevant event occurs and decremented when a thread (e.g., a user-space device driver) indicates via the **evc_wait** trap that it wishes to process an event. Each task refers to an event by referencing its event counter. The appropriate event counter is communicated to a thread in a driver-specific way.[7] The expression $EVENT\_COUNTER$ denotes the set of all event counters.

Mach Definition 96

$$[EVENT\_COUNTER]$$

Each event counter may have at most one thread, denoted by $\underline{t}hread\_waiting(evc)$, waiting for it. Furthermore, each thread may be waiting on at most one event counter. The number of event that are queued and waiting to be processed by a thread is denoted by $\underline{e}vent\_count(evc)$. The expression $\underline{s}upplying\_device$ denotes the kernel-space device driver that supplied the event counter.

Mach Definition 97

$$\begin{array}{l} \textit{Events} \\ \hline ThreadExecStatus \\ \underline{t}hread\_waiting : EVENT\_COUNTER \rightarrowtail\!\!\!\!\!\!+\ THREAD \\ \underline{e}vent\_count : EVENT\_COUNTER \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{Z} \\ \underline{s}upplying\_device : EVENT\_COUNTER \mathrel{+\!\!\!\!\!\rightarrow} DEVICE \\ \hline \mathrm{dom}\ \underline{e}vent\_count = \mathrm{dom}\ \underline{s}upplying\_device \\ \mathrm{dom}\ \underline{t}hread\_waiting \subseteq \mathrm{dom}\ \underline{e}vent\_count \\ \mathrm{ran}\ \underline{t}hread\_waiting \subseteq \\ \qquad \{thread : THREAD \mid thread \in \underline{t}hread\_exists \land Waiting \in \underline{r}un\_state(thread)\} \\ \hline \end{array}$$

Devices can be associated with memory objects that can then be mapped into address spaces. We use $\underline{m}apped\_devices$ to denote the set of devices that have been associated with memory objects.

Mach Definition 98

$$\begin{array}{l} \textit{MappedDevices} \\ \hline \underline{m}apped\_devices : \mathbb{P}\ DEVICE \\ \hline \end{array}$$

Each device has two associated queues of data records. We use $\underline{d}evice\_in(dev)$ and $\underline{d}evice\_out(dev)$ to denote, respectively, data input and output through the device. Data read from dev is dequeued from $\underline{d}evice\_in(dev)$, and data written to dev is enqueued to $\underline{d}evice\_out(dev)$.

[7] Threads may also wait for events that occur while the system is operating in kernel space (e.g., another thread becomes suspended). This is handled through a separate waiting mechanism that is not modeled in the FTLS.

Mach Definition 99

$[DEVICE\_RECORD]$

Mach Definition 100

$$\begin{array}{l} \textit{DeviceData} \\ \hline \underline{d}evice\_in : DEVICE \longrightarrow \mathrm{seq}\, DEVICE\_RECORD \\ \underline{d}evice\_out : DEVICE \longrightarrow \mathrm{seq}\, DEVICE\_RECORD \\ \hline \end{array}$$

Each device can have associated filters that are used to route data received through the device. Each filter has an associated port to which data accepted by the filter is delivered. Furthermore, a priority can be associated with each port to indicate the ordering when there are multiple ports associated with the filter. We use $\underline{d}evice\_filter\_info(dev)$ to indicate the set of $(device\_filter, port, filter\_priority)$ triples associated with dev.

Mach Definition 101

$[DEVICE\_FILTER, FILTER\_PRIORITY]$
$DEVICE\_FILTER\_INFO == DEVICE\_FILTER \times PORT \times FILTER\_PRIORITY$

Mach Definition 102

$$\begin{array}{l} \textit{DeviceFilterInfo} \\ \hline \underline{d}evice\_filter\_info : DEVICE \longrightarrow \mathbb{P}\, DEVICE\_FILTER\_INFO \\ \hline \end{array}$$

Each device has an associated status. We use $\underline{d}evice\_status(dev)$ to denote dev's status.

Mach Definition 103

$[DEVICE\_STATUS]$

Mach Definition 104

$$\begin{array}{l} \textit{DeviceStatus} \\ \hline \underline{d}evice\_status : DEVICE \longrightarrow DEVICE\_STATUS \\ \hline \end{array}$$

Mach Definition 105

$$\begin{array}{l} \textit{Devices} \\ \hline DeviceOpenCount \\ Events \\ MappedDevices \\ DeviceData \\ DeviceFilterInfo \\ DeviceStatus \\ \hline \end{array}$$

4.14 Summary

The data structures defined in the previous sections comprise the Mach system state. The type $Mach$ is used to denote the set of Mach system states.

Mach Definition 106

$$
\begin{aligned}
&Process \mathrel{\widehat=} Threads \land TaskPriority \land TaskSuspendCount\\
&\quad \land EmulationVector\\
&Ipc \mathrel{\widehat=} PortNameSpace \land RegisteredRights \land Notifications\\
&\quad \land PortSummary \land PortClasses \land Messages\\
&Processor \mathrel{\widehat=} HostsAndProcessors \land ProcessorAndProcessorSet \land TaskAndProcessorSet\\
&\quad \land ThreadAndProcessorSet \land ThreadsAndProcessors
\end{aligned}
$$

Mach Definition 107

$$
\begin{array}{|l}
\hline
\textit{Mach}\\
Exist\\
Process\\
Ipc\\
Processor\\
MemorySystem\\
HostTime\\
Devices\\
\hline
\underline{m}anager = receiver \circ object\_port\\
\hline
\end{array}
$$

Section 5
DTOS State Extensions

This section describes extensions to the base Mach microkernel state that are needed to support the DTOS kernel. The DTOS kernel is intended to support a wide range of policies. Thus, the state components described in this section are independent of any specific access control policy.

In general, an access control policy consists of three components. First, security attributes must be associated with the subjects accessing entities in the system. Second, security attributes must be associated with the entities in the system that subjects access. Finally, a rule must be defined that indicates the set of accesses that a subject with a given attribute can make to an entity with a given attribute. To provide policy flexibility, the DTOS kernel abstracts the security attributes associated with specific policies into sets of *security identifiers*. Although the kernel relies upon a security server to define the policy to be enforced, the kernel maintains a cache of accesses previously authorized by the security server.

In addition to providing a framework for access control policies, the DTOS kernel also enhances the security of the Mach IPC mechanism.

The organization of this section is as follows:

- Section 5.1, **Subject Security Information**, describes the security information recorded for subjects.
- Section 5.2, **Object Security Information**, describes the security information recorded for objects.
- Section 5.3, **Security Identifiers for Access Computations**, describes some security identifiers used only in access computations.
- Section 5.4, **Permissions**, describes the permissions enforced in DTOS.
- Section 5.5, **Access Vector Cache**, describes the DTOS kernel's access vector cache.
- Section 5.6, **Message Security Information**, describes the security information associated with messages to enhance the security of the Mach IPC mechanism.
- Section 5.7, **Task Creation Information**, describes information associated with tasks to enhance the security of the Mach approach for process initiation.
- Section 5.8, **Server Ports**, describes ports used by the kernel for communication with other servers.
- Section 5.9, **Memory Region Protections**, describes information associated with regions to allow the DTOS kernel to enforce access.

5.1 Subject Security Information

Subjects in DTOS are threads executing within tasks. Each task has a *subject security identifier* (SSI). The set SSI denotes the set of all SSIs.

We will occasionally need to identify two distinct components of each SID, a *mandatory* security identifier (MID) and an *authentication* identifier (AID). We use the types MID and AID to denote, respectively, MIDs and AIDs. The functions Ssi_to_mid and Ssi_to_aid are used to map SSIs to MIDs and AIDs.

DTOS Kernel Definition 1

$[SSI]$

$[MID, AID]$

$$
\begin{array}{|l}
Ssi\_to\_mid : SSI \longrightarrow MID \\
Ssi\_to\_aid : SSI \longrightarrow AID
\end{array}
$$

The expressions $\underline{t}ask\_sid(task)$, $task\_mid(task)$ and $task\_aid(task)$ are used to denote the SSI, MID and AID associated with a task. The expression $thread\_sid(thread)$ denotes the SSI associated with a thread. It is defined to be the SSI of its parent task.

DTOS Kernel Definition 2

$$
\begin{array}{|l}
\hline
\textit{SubjectSid} \\
TaskExist \\
ThreadExist \\
TasksAndThreads \\
\underline{t}ask\_sid : TASK \leftrightarrow SSI \\
task\_mid : TASK \leftrightarrow MID \\
task\_aid : TASK \leftrightarrow AID \\
thread\_sid : THREAD \leftrightarrow SSI \\
\hline
\operatorname{dom} \underline{t}ask\_sid = \operatorname{dom} task\_mid = \operatorname{dom} task\_aid = \underline{t}ask\_exists \\
\operatorname{dom} thread\_sid = \underline{t}hread\_exists \\
task\_mid = Ssi\_to\_mid \circ \underline{t}ask\_sid \\
task\_aid = Ssi\_to\_aid \circ \underline{t}ask\_sid \\
thread\_sid = \underline{t}ask\_sid \circ owning\_task \\
\hline
\end{array}
$$

5.2 Object Security Information

Each port has an associated *object security identifier* (OSI) that represents the security attributes associated with the port. Similarly, each memory region has an associated OSI. The set OSI denotes the set of all OSIs.

The functions Osi_to_mid and Osi_to_aid are used to map OSIs to MIDs and AIDs.

DTOS Kernel Definition 3

$[OSI]$

$$
\begin{array}{|l}
Osi\_to\_mid : OSI \longrightarrow MID \\
Osi\_to\_aid : OSI \longrightarrow AID
\end{array}
$$

The expressions $\underline{p}ort\_sid(port)$, $port\_mid(port)$ and $port\_aid(port)$ are used to denote the OSI, MID and AID associated with a port.

DTOS Kernel Definition 4

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PortSid \quad}} \\
PortExist \\
\underline{p}ort\_sid : PORT \rightarrowtail OSI \\
port\_mid : PORT \rightarrowtail MID \\
port\_aid : PORT \rightarrowtail AID \\
\hline
\mathrm{dom}\, \underline{p}ort\_sid = \mathrm{dom}\, port\_mid = \mathrm{dom}\, port\_aid = \underline{p}ort\_exists \\
port\_mid = Osi\_to\_mid \circ \underline{p}ort\_sid \\
port\_aid = Osi\_to\_aid \circ \underline{p}ort\_sid \\
\hline
\end{array}
$$

Each task and thread has a self port on which the kernel receives requests to perform an action on the task or thread. The OSI of the self ports is derived from the SSI of the corresponding task. The expressions $Task\_port\_sid(ssi)$ and $Thread\_port\_sid(ssi)$ indicate the corresponding OSIs. When memory is allocated, it is labeled with an OSI that is derived from the SSI of the owning task. The expression $Default\_vm\_port\_sid(ssi)$ indicates the derived OSI. Similarly, when a port is created, it is labeled with an OSI derived from the SSI of the task in whose IPC name space it is allocated. The expression $Default\_port\_sid(ssi)$ indicates the derived OSI.

DTOS Kernel Definition 5

$$
\begin{array}{|l}
Task\_port\_sid : SSI \longrightarrow OSI \\
Thread\_port\_sid : SSI \longrightarrow OSI \\
Default\_vm\_port\_sid : SSI \longrightarrow OSI \\
Default\_port\_sid : SSI \longrightarrow OSI \\
\hline
\mathsf{disjoint}\ \langle \mathrm{ran}\ Task\_port\_sid, \mathrm{ran}\ Thread\_port\_sid, \\
\quad \mathrm{ran}\ Default\_vm\_port\_sid, \mathrm{ran}\ Default\_port\_sid \rangle
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad KernelPortSid \quad}} \\
TasksAndThreads \\
SpecialPurposePorts \\
SubjectSid \\
PortSid \\
\hline
\forall\, task : \underline{t}ask\_exists \\
\bullet\ \underline{p}ort\_sid(task\_self(task)) = Task\_port\_sid(\underline{t}ask\_sid(task)) \\
\forall\, thread : \underline{t}hread\_exists \\
\bullet\ \underline{p}ort\_sid(thread\_self(thread)) = Thread\_port\_sid(\underline{t}ask\_sid(owning\_task(thread))) \\
\hline
\end{array}
$$

The expressions $\underline{p}age\_sid(task, page\_index)$, $page\_mid(task, page\_index)$ and $page\_aid(task, page\_index)$ are used to denote the OSI, MID and AID associated with a page. Note that $\underline{p}age\_sid$ effectively associates an OSI with each allocated address in a task's address space. If a page is managed and the manager is not the default memory manager, then the SID of the page is derived from the SID of the pager port of the object containing the page. The derivation of page SIDs from pager port SIDs is modeled by the function $Pp\_to\_page\_sid$.

DTOS Kernel Definition 6

$$
\begin{array}{|l}
Pp\_to\_page\_sid : OSI \rightarrowtail OSI
\end{array}
$$

DTOS Kernel Definition 7

$$
\begin{array}{|l}
\hline
\textit{PageSid} \\
\textit{AddressSpace} \\
\textit{Memory} \\
\textit{TasksAndPorts} \\
\textit{PortSid} \\
\underline{p}age\_sid : TASK \times PAGE\_INDEX \pfun OSI \\
page\_mid : TASK \times PAGE\_INDEX \pfun MID \\
page\_aid : TASK \times PAGE\_INDEX \pfun AID \\
\hline
\mathrm{dom}\, \underline{p}age\_sid = \mathrm{dom}\, page\_mid = \mathrm{dom}\, page\_aid = \underline{a}llocated \\
page\_mid = Osi\_to\_mid \circ \underline{p}age\_sid \\
page\_aid = Osi\_to\_aid \circ \underline{p}age\_sid \\
\\
(\forall\, task\_va\_pair : TASK \times PAGE\_INDEX;\ memory : MEMORY; \\
\quad port : PORT \\
\mid (task\_va\_pair, memory) \in mapped\_memory \\
\quad \land (memory, port) \in object\_port \\
\quad \land receiver(port) \neq receiver(\underline{d}efault\_mem\_manager) \\
\bullet\ \underline{p}age\_sid(task\_va\_pair) = Pp\_to\_page\_sid(\underline{p}ort\_sid(port))) \\
\hline
\end{array}
$$

Editorial Note:
Need to figure out if their is a better way to check that the memory is not being paged by the default memory manager.

DTOS Kernel Definition 8

$$
\begin{array}{|l}
\hline
\textit{ObjectSid} \\
\textit{PortSid} \\
\textit{KernelPortSid} \\
\textit{PageSid} \\
\hline
\mathrm{dom}\, Pp\_to\_page\_sid \subseteq \mathrm{ran}\, \underline{p}ort\_sid \\
\mathrm{ran}\, Pp\_to\_page\_sid \subseteq \mathrm{ran}\, \underline{p}age\_sid \\
\hline
\end{array}
$$

5.3 Security Identifiers for Access Computations

Access computations in the DTOS kernel are generally made based upon the SSI of the task accessing an object and the OSI of the accessed object. This section discusses a few special cases in which other security identifiers are used.

Sometimes kernel requests can have side effects resulting in outcalls from the kernel, for instance, to deliver dead name notifications. For fine grained control over such operations it is desirable to distinguish between the kernel sending such a message to a port as a side effect of another request and the client directly sending a message to the port. To provide for this, such side effects are sometimes controlled based not upon the SSI of the client but upon an SSI derived from the client's SSI and indicating that it is the kernel acting on behalf of a client with the given SSI. The function $Derive\_kernel\_as$ maps an SSI s_1 to the derived SSI s_2 representing the kernel acting on behalf of a task with SSI s_1. We use $kernel\_as(task)$ to denote the derived SSI indicating the kernel acting on behalf of a task $task$.

DTOS Kernel Definition 9

$$Derive\_kernel\_as : SSI \longrightarrow SSI$$

$KernelAs$

$SubjectSid$

$kernel\_as : TASK \longrightarrow SSI$

$$kernel\_as = \underline{t}ask\_sid \fatsemi Derive\_kernel\_as$$

One of the features of Mach is that it allows tasks to perform operations on other tasks that have not traditionally been provided by operating systems. For example, Mach allows tasks to access memory regions in other tasks while one of the features of traditional operating systems is the separation of address spaces. To provide finer control over task accesses, we define $Task\_self\_sid$ to be a value to be used in access computations governing accesses a task makes to itself. Similarly, we use $Thread\_self\_sid$ to be a value to be used in access computations governing accesses a task makes to threads that it owns. The security policy should normally be defined in such a way as to prevent any kernel entities from being assigned $Task\_self\_sid$ or $Thread\_self\_sid$ as their SID.[8] Instead, these SIDs indicate to security servers that the kernel requires an access computation to be performed between a task and the task itself or between a task and one of the task's threads. One potential use of this finer control would be to contain a faulty task by preventing it from corrupting other tasks having the same SID.

We define $task\_target(task_1, task_2)$ to be the OSI of $task_2$'s self port if $task_1$ and $task_2$ are different and $Task\_self\_sid$, otherwise. Analogously, we define $thread\_target(task, thread)$ to be the OSI of $thread$'s self port if $thread$ does not belong to $task$ and $Thread\_self\_sid$, otherwise. When $task_1$ attempts to operate on $task_2$, the kernel enforces accesses on the pair $(\underline{t}ask\_sid(task_1), task\_target(task_1, task_2))$. Analogously, operations that $task$ performs on $thread$ are governed by the accesses recorded for $(\underline{t}ask\_sid(task), thread\_target(task, thread))$. This allows separate permissions sets to be applied when a task operates on itself versus operating on another process with the same SSI.

DTOS Kernel Definition 10

$Task\_self\_sid : OSI$

$Thread\_self\_sid : OSI$

$$Task\_self\_sid \neq Thread\_self\_sid$$

DTOS Kernel Definition 11

[8] This property is not guaranteed by the kernel. For example, a **mach_port_allocate_secure** request may specify a self SID as the SID for the newly created port. If the security server allows the client to add a name to the target task and allows the target task to hold a receive right for a port with the specified SID, the request will succeed and the port will be labeled with a self SID.

$$
\begin{array}{l}
\textbf{TargetSids} \\
PortSid \\
TasksAndThreads \\
SpecialPurposePorts \\
task\_target : TASK \times TASK \rightarrowtail\!\!\!\!\!\!+\; OSI \\
thread\_target : TASK \times THREAD \rightarrowtail\!\!\!\!\!\!+\; OSI \\
\hline
\{Task\_self\_sid, Thread\_self\_sid\} \cap \operatorname{ran} \underline{p}ort\_sid = \varnothing \\
\operatorname{dom} task\_target = TASK \times \underline{t}ask\_exists \\
\operatorname{dom} thread\_target = TASK \times \underline{t}hread\_exists \\
\forall\, task_1, task_2 : TASK \\
\bullet\; task\_target(task_1, task_2) \\
\quad = \textbf{if } task_1 = task_2 \textbf{ then } Task\_self\_sid \\
\quad \textbf{else } \underline{p}ort\_sid(task\_self(task_2)) \\
\forall\, task : TASK; thread : THREAD \\
\bullet\; thread\_target(task, thread) \\
\quad = \textbf{if } task = owning\_task(thread) \textbf{ then } Thread\_self\_sid \\
\quad \textbf{else } \underline{p}ort\_sid(thread\_self(thread))
\end{array}
$$

Editorial Note:
In the prototype $Task\_self\_sid$ and $Thread\_self\_sid$ are not implemented as constants. Rather, they are derived from the corresponding subject SID in the same way as the derived SIDs $Task\_port\_sid$, $Thread\_port\_sid$, $Default\_vm\_port\_sid$ and $Default\_port\_sid$ which are described above. Given the way the self SIDs are used the two approaches are equivalent.

5.4 Permissions

The DTOS security policy constrains when clients may obtain *services*. The security policy is enforced by:

- associating a set of allowed permissions[9] with each SSI-OSI pair,
- associating a set of required permissions with each service, and
- granting service only when the required permissions are contained in the allowed permissions for the client to the target for the operation.

The set $PERMISSION$ denotes the set of all permissions. This set contains permissions governing kernel services as well as permissions governing services provided by user space servers.

The set $Kernel\_permission$ is used to denote the subset of $PERMISSION$ that governs kernel services.

DTOS Kernel Definition 12

$$[PERMISSION]$$

$$\mid\; Kernel\_permission : \mathbb{P}\, PERMISSION$$

[9] Note that the terms *access vector*, *service vector*, and *permission set* are used somewhat interchangeably.

The elements of $Kernel\_permission$ are enumerated in subsections 5.4.1-5.4.14. The operator $Values\_partition$ is formally defined in Appendix B. Informally, the expression $\langle val_1, \ldots, val_n \rangle \ Values\_partition \ S$ denotes that the values $val_1, \ldots, val_n$ are unique values that together comprise the set val_set.

5.4.1 IPC Permissions

The DTOS kernel enforces the following "IPC" permissions: $Can\_receive$, Can_send, $Hold\_receive$, $Hold\_send$, $Hold\_send\_once$, $Interpose$, Map_vm_region, Set_reply, $Specify$, $Transfer\_ool$, $Transfer\_receive$, $Transfer\_rights$, $Transfer\_send$, $Transfer\_send\_once$. We use $Ipc\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 13

$$
\begin{array}{l}
Ipc\_permissions : \mathbb{P}\ PERMISSION \\
Can\_receive, Can\_send, Hold\_receive, \\
Hold\_send, Hold\_send\_once, Interpose, \\
Map\_vm\_region, Set\_reply, Specify, \\
Transfer\_ool, Transfer\_receive, Transfer\_rights, \\
Transfer\_send, Transfer\_send\_once : \\
\quad PERMISSION \\
\hline
\langle Can\_receive, Can\_send, Hold\_receive, Hold\_send, Hold\_send\_once, Interpose, \\
\qquad Map\_vm\_region, Set\_reply, Specify, Transfer\_ool, Transfer\_receive, \\
\qquad Transfer\_rights, Transfer\_send, Transfer\_send\_once \rangle \\
\quad Values\_partition\ Ipc\_permissions
\end{array}
$$

5.4.2 Port Permissions

The DTOS kernel enforces the following permissions on port requests: Add_name, $Alter\_pns\_info$, $Extract\_right$, $Lookup\_ports$, $Manipulate\_port\_set$, $Observe\_pns\_info$, $Port\_rename$, $Register\_notification$, $Register\_ports$, $Remove\_name$. We use $Port\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 14

$$
\begin{array}{l}
Port\_permissions : \mathbb{P}\ PERMISSION \\
Add\_name, Alter\_pns\_info, Extract\_right, \\
Lookup\_ports, Manipulate\_port\_set, Observe\_pns\_info, \\
Port\_rename, Register\_notification, Register\_ports, \\
Remove\_name : \\
\quad PERMISSION \\
\hline
\langle Add\_name, Alter\_pns\_info, Extract\_right, Lookup\_ports, \\
\qquad Manipulate\_port\_set, Observe\_pns\_info, Port\_rename, \\
\qquad Register\_notification, Register\_ports, Remove\_name \rangle \\
\quad Values\_partition\ Port\_permissions
\end{array}
$$

5.4.3 VM Permissions

The DTOS kernel enforces the following permissions on VM requests: $Access\_machine\_attribute$, $Allocate\_vm\_region$, $Chg\_vm\_region\_prot$, $Copy\_vm$, $Deallocate\_vm\_region$, $Get\_vm\_region\_info$, $Get\_vm\_statistics$, $Read\_vm\_region$, $Set\_vm\_region\_inherit$, $Wire\_vm\_for\_task$, $Write\_vm\_region$. We use $Vm\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 15

$$
\begin{array}{|l}
Vm\_permissions : \mathbb{P}\ PERMISSION \\
Access\_machine\_attribute, Allocate\_vm\_region, Chg\_vm\_region\_prot, \\
Copy\_vm, Deallocate\_vm\_region, Get\_vm\_region\_info, \\
Get\_vm\_statistics, Read\_vm\_region, Set\_vm\_region\_inherit, \\
Wire\_vm\_for\_task, Write\_vm\_region : \\
\quad PERMISSION \\
\hline
\langle Access\_machine\_attribute, Allocate\_vm\_region, Chg\_vm\_region\_prot, \\
\qquad Copy\_vm, Deallocate\_vm\_region, Get\_vm\_region\_info, Get\_vm\_statistics, \\
\qquad Read\_vm\_region, Set\_vm\_region\_inherit, Wire\_vm\_for\_task, \\
\qquad Write\_vm\_region\rangle \\
\quad Values\_partition\ Vm\_permissions
\end{array}
$$

5.4.4 Memory Object Permissions

The DTOS kernel enforces the following permissions on memory requests: $Have\_execute$, $Have\_read$, $Have\_write$, $Page\_vm\_region$. We use $Memory\_object\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 16

$$
\begin{array}{|l}
Memory\_object\_permissions : \mathbb{P}\ PERMISSION \\
Have\_execute, Have\_read, Have\_write, \\
Page\_vm\_region : \\
\quad PERMISSION \\
\hline
\langle Have\_execute, Have\_read, Have\_write, Page\_vm\_region\rangle \\
\quad Values\_partition\ Memory\_object\_permissions
\end{array}
$$

5.4.5 Pager Permissions

The DTOS kernel enforces the following permissions on pager requests: $Change\_page\_locks$, $Destroy\_object$, $Get\_attributes$, $Invoke\_lock\_request$, $Make\_page\_precious$, $Provide\_data$, $Remove\_page$, $Revoke\_ibac$, $Save\_page$, $Set\_attributes$, Set_ibac_port, $Supply\_ibac$. We use $Pager\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 17

$Pager\_permissions : \mathbb{P}\ PERMISSION$
$Change\_page\_locks, Destroy\_object, Get\_attributes,$
$Invoke\_lock\_request, Make\_page\_precious, Provide\_data,$
$Remove\_page, Revoke\_ibac, Save\_page,$
$Set\_attributes, Set\_ibac\_port, Supply\_ibac :$
$PERMISSION$

$\langle Change\_page\_locks, Destroy\_object, Get\_attributes, Invoke\_lock\_request,$
$Make\_page\_precious, Provide\_data, Remove\_page, Revoke\_ibac, Save\_page,$
$Set\_attributes, Set\_ibac\_port, Supply\_ibac\rangle$
$Values\_partition\ Pager\_permissions$

5.4.6 Thread Permissions

The DTOS kernel enforces the following permissions on thread requests: $Abort\_thread$, $Abort\_thread\_depress$, $Assign\_thread\_to\_pset$, Can_swtch, Can_swtch_pri, $Depress\_pri$, $Get\_thread\_assignment$, $Get\_thread\_exception\_port$, Get_thread_info, $Get\_thread\_kernel\_port$, Get_thread_state, $Initiate\_secure$, $Raise\_exception$, $Resume\_thread$, $Sample\_thread$, $Set\_max\_thread\_priority$, $Set\_thread\_exception\_port$, $Set\_thread\_kernel\_port$, Set_thread_policy, $Set\_thread\_priority$, Set_thread_state, $Suspend\_thread$, $Switch\_thread$, $Terminate\_thread$, $Wait\_evc$, $Wire\_thread\_into\_memory$. We use $Thread\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 18

$Thread\_permissions : \mathbb{P}\ PERMISSION$
$Abort\_thread, Abort\_thread\_depress, Assign\_thread\_to\_pset,$
$Can\_swtch, Can\_swtch\_pri, Depress\_pri,$
$Get\_thread\_assignment, Get\_thread\_exception\_port, Get\_thread\_info,$
$Get\_thread\_kernel\_port, Get\_thread\_state, Initiate\_secure,$
$Raise\_exception, Resume\_thread, Sample\_thread,$
$Set\_max\_thread\_priority, Set\_thread\_exception\_port, Set\_thread\_kernel\_port,$
$Set\_thread\_policy, Set\_thread\_priority, Set\_thread\_state,$
$Suspend\_thread, Switch\_thread, Terminate\_thread,$
$Wait\_evc, Wire\_thread\_into\_memory :$
$PERMISSION$

$\langle Abort\_thread, Abort\_thread\_depress, Assign\_thread\_to\_pset, Can\_swtch,$
$Can\_swtch\_pri, Depress\_pri, Get\_thread\_assignment,$
$Get\_thread\_exception\_port, Get\_thread\_info, Get\_thread\_kernel\_port,$
$Get\_thread\_state, Initiate\_secure, Raise\_exception, Resume\_thread,$
$Sample\_thread, Set\_max\_thread\_priority, Set\_thread\_exception\_port,$
$Set\_thread\_kernel\_port, Set\_thread\_policy, Set\_thread\_priority,$
$Set\_thread\_state, Suspend\_thread, Switch\_thread, Terminate\_thread,$
$Wait\_evc, Wire\_thread\_into\_memory\rangle$
$Values\_partition\ Thread\_permissions$

5.4.7 Task Permissions

The DTOS kernel enforces the following permissions on task requests: Add_thread, Add_thread_secure, $Assign\_task\_to\_pset$, $Change\_sid$, $Chg\_task\_priority$, $Create\_task$, $Create\_task\_secure$, $Cross\_context\_create$, $Cross\_context\_inherit$, $Get\_emulation$, $Get\_task\_assignment$, $Get\_task\_boot\_port$, $Get\_task\_exception\_port$, Get_task_info, $Get\_task\_kernel\_port$, $Get\_task\_threads$, $Make\_sid$, $Resume\_task$, $Sample\_task$, $Set\_emulation$, Set_ras, $Set\_task\_boot\_port$, $Set\_task\_exception\_port$, $Set\_task\_kernel\_port$, $Suspend\_task$, $Terminate\_task$, $Transition\_sid$. We use $Task\_task\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 19

$$
\begin{array}{l}
Task\_task\_permissions : \mathbb{P}\ PERMISSION \\
Add\_thread, Add\_thread\_secure, Assign\_task\_to\_pset, \\
Change\_sid, Chg\_task\_priority, Create\_task, \\
Create\_task\_secure, Cross\_context\_create, Cross\_context\_inherit, \\
Get\_emulation, Get\_task\_assignment, Get\_task\_boot\_port, \\
Get\_task\_exception\_port, Get\_task\_info, Get\_task\_kernel\_port, \\
Get\_task\_threads, Make\_sid, Resume\_task, \\
Sample\_task, Set\_emulation, Set\_ras, \\
Set\_task\_boot\_port, Set\_task\_exception\_port, Set\_task\_kernel\_port, \\
Suspend\_task, Terminate\_task, Transition\_sid : \\
\quad PERMISSION \\
\hline
\langle Add\_thread, Add\_thread\_secure, Assign\_task\_to\_pset, Change\_sid, \\
\qquad Chg\_task\_priority, Create\_task, Create\_task\_secure, \\
\qquad Cross\_context\_create, Cross\_context\_inherit, Get\_emulation, \\
\qquad Get\_task\_assignment, Get\_task\_boot\_port, Get\_task\_exception\_port, \\
\qquad Get\_task\_info, Get\_task\_kernel\_port, Get\_task\_threads, Make\_sid, \\
\qquad Resume\_task, Sample\_task, Set\_emulation, Set\_ras, Set\_task\_boot\_port, \\
\qquad Set\_task\_exception\_port, Set\_task\_kernel\_port, Suspend\_task, \\
\qquad Terminate\_task, Transition\_sid \rangle \\
\quad Values\_partition\ Task\_task\_permissions
\end{array}
$$

We use $Task\_permissions$ to denote the union of $Task\_task\_permissions$, $Port\_permissions$, and $Vm\_permissions$.

DTOS Kernel Definition 20

$$
\begin{array}{l}
Task\_permissions : \mathbb{P}\ PERMISSION \\
\hline
\langle Port\_permissions, Vm\_permissions, Task\_task\_permissions \rangle \text{ partition } Task\_permissions
\end{array}
$$

5.4.8 Host Name Port Permissions

The DTOS kernel enforces the following permissions on host name port requests: $Create\_pset$, $Flush\_permission$, Get_audit_port, $Get\_authentication\_port$, Get_crypto_port, $Get\_default\_pset\_name$, $Get\_host\_control\_port$, Get_host_info, Get_host_name, $Get\_host\_version$, $Get\_negotiation\_port$, $Get\_network\_ss\_port$, $Get\_security\_master\_port$, $Get\_security\_client\_port$, $Get\_special\_port$, Get_time, $Pset\_names$, Set_audit_port, $Set\_authentication\_port$, Set_crypto_port, $Set\_negotiation\_port$, $Set\_network\_ss\_port$, $Set\_security\_master\_port$, $Set\_security\_client\_port$, $Set\_special\_port$. We use $Host\_name\_port\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 21

$$
\begin{array}{|l}
Host\_name\_port\_permissions : \mathbb{P}\, PERMISSION \\
Create\_pset, Flush\_permission, Get\_audit\_port, \\
Get\_authentication\_port, Get\_crypto\_port, Get\_default\_pset\_name, \\
Get\_host\_control\_port, Get\_host\_info, Get\_host\_name, \\
Get\_host\_version, Get\_negotiation\_port, Get\_network\_ss\_port, \\
Get\_security\_master\_port, Get\_security\_client\_port, Get\_special\_port, \\
Get\_time, Pset\_names, Set\_audit\_port, \\
Set\_authentication\_port, Set\_crypto\_port, Set\_negotiation\_port, \\
Set\_network\_ss\_port, Set\_security\_master\_port, Set\_security\_client\_port, \\
Set\_special\_port : \\
\quad PERMISSION \\
\hline
\langle Create\_pset, Flush\_permission, Get\_audit\_port, \\
\qquad Get\_authentication\_port, Get\_crypto\_port, Get\_default\_pset\_name, \\
\qquad Get\_host\_control\_port, Get\_host\_info, Get\_host\_name, Get\_host\_version, \\
\qquad Get\_negotiation\_port, Get\_network\_ss\_port, Get\_security\_master\_port, \\
\qquad Get\_security\_client\_port, Get\_special\_port, Get\_time, \\
\qquad Pset\_names, Set\_audit\_port, Set\_authentication\_port, \\
\qquad Set\_crypto\_port, Set\_negotiation\_port, Set\_network\_ss\_port, \\
\qquad Set\_security\_master\_port, Set\_security\_client\_port, Set\_special\_port\rangle \\
\quad Values\_partition\ Host\_name\_port\_permissions
\end{array}
$$

5.4.9 Host Control Port Permissions

The DTOS kernel enforces the following permissions on host control port requests: Get_boot_info, $Get\_host\_processors$, $Pset\_ctrl\_port$, $Reboot\_host$, $Set\_default\_memory\_mgr$, Set_time, $Wire\_thread$, $Wire\_vm$. We use $Host\_control\_port\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 22

$$
\begin{array}{|l}
Host\_control\_port\_permissions : \mathbb{P}\, PERMISSION \\
Get\_boot\_info, Get\_host\_processors, Pset\_ctrl\_port, \\
Reboot\_host, Set\_default\_memory\_mgr, Set\_time, \\
Wire\_thread, Wire\_vm : \\
\quad PERMISSION \\
\hline
\langle Get\_boot\_info, Get\_host\_processors, Pset\_ctrl\_port, Reboot\_host, \\
\qquad Set\_default\_memory\_mgr, Set\_time, Wire\_thread, Wire\_vm\rangle \\
\quad Values\_partition\ Host\_control\_port\_permissions
\end{array}
$$

5.4.10 Processor Permissions

The DTOS kernel enforces the following permissions on processor requests: $Assign\_processor\_to\_set$, $Get\_processor\_assignment$, $Get\_processor\_info$, $May\_control\_processor$. We use $Processor\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 23

$$\begin{array}{l} Processor\_permissions : \mathbb{P}\ PERMISSION \\ Assign\_processor\_to\_set, Get\_processor\_assignment, Get\_processor\_info, \\ May\_control\_processor : \\ \quad PERMISSION \\ \hline \langle Assign\_processor\_to\_set, Get\_processor\_assignment, Get\_processor\_info, \\ \qquad May\_control\_processor\rangle \\ \quad Values\_partition\ Processor\_permissions \end{array}$$

5.4.11 Processor Set Name Port Permissions

The DTOS kernel enforces the following permissions on processor set name port requests: Get_pset_info. We use $Procset\_name\_port\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 24

$$\begin{array}{l} Procset\_name\_port\_permissions : \mathbb{P}\ PERMISSION \\ Get\_pset\_info : \\ \quad PERMISSION \\ \hline \langle Get\_pset\_info\rangle \\ \quad Values\_partition\ Procset\_name\_port\_permissions \end{array}$$

5.4.12 Processor Set Control Port Permissions

The DTOS kernel enforces the following permissions on processor set control port requests: $Assign\_processor$, $Assign\_task$, $Assign\_thread$, $Chg\_pset\_max\_pri$, $Define\_new\_scheduling\_policy$, $Destroy\_pset$, $Invalidate\_scheduling\_policy$, $Observe\_pset\_processes$. We use $Procset\_control\_port\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 25

$$\begin{array}{l} Procset\_control\_port\_permissions : \mathbb{P}\ PERMISSION \\ Assign\_processor, Assign\_task, Assign\_thread, \\ Chg\_pset\_max\_pri, Define\_new\_scheduling\_policy, Destroy\_pset, \\ Invalidate\_scheduling\_policy, Observe\_pset\_processes : \\ \quad PERMISSION \\ \hline \langle Assign\_processor, Assign\_task, Assign\_thread, Chg\_pset\_max\_pri, \\ \qquad Define\_new\_scheduling\_policy, Destroy\_pset, \\ \qquad Invalidate\_scheduling\_policy, Observe\_pset\_processes\rangle \\ \quad Values\_partition\ Procset\_control\_port\_permissions \end{array}$$

We use $Procset\_permissions$ to denote the union of $Procset\_name\_port\_permissions$ and $Procset\_control\_port\_permissions$.

DTOS Kernel Definition 26

$$\begin{array}{l} Procset\_permissions : \mathbb{P}\ PERMISSION \\ \hline \langle Procset\_name\_port\_permissions, Procset\_control\_port\_permissions\rangle \\ \quad \mathsf{partition}\ Procset\_permissions \end{array}$$

5.4.13 Device Permissions

The DTOS kernel enforces the following permissions on device requests: $Close\_device$, $Control\_pager$, Get_device_status, Map_device, $Open\_device$, $Read\_device$, Set_device_filter, Set_device_status, $Write\_device$. We use $Device\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 27

$$
\begin{array}{l}
Device\_permissions : \mathbb{P}\ PERMISSION \\
Close\_device, Control\_pager, Get\_device\_status, \\
Map\_device, Open\_device, Read\_device, \\
Set\_device\_filter, Set\_device\_status, Write\_device : \\
\quad PERMISSION \\
\hline
\langle Close\_device, Control\_pager, Get\_device\_status, Map\_device, Open\_device, \\
\qquad Read\_device, Set\_device\_filter, Set\_device\_status, Write\_device \rangle \\
\quad Values\_partition\ Device\_permissions
\end{array}
$$

5.4.14 Kernel Reply Port Permissions

The DTOS kernel enforces the following permissions on requests sent to kernel reply ports: $Provide\_permission$. We use $Kernel\_reply\_permissions$ to denote this set of permissions.

DTOS Kernel Definition 28

$$
\begin{array}{l}
Kernel\_reply\_permissions : \mathbb{P}\ PERMISSION \\
Provide\_permission : \\
\quad PERMISSION \\
\hline
\langle Provide\_permission \rangle \\
\quad Values\_partition\ Kernel\_reply\_permissions
\end{array}
$$

We do not require that all of the above sets of permissions be non-overlapping. The only such requirement is that the $Ipc\_permissions$ do not overlap with any of the other sets. This is consistent with the current prototype in which permissions are simply integers specifying positions in access vectors. Because there are different types of access vector depending upon the type of target object, multiple permissions may specify the same access vector position. Every vector contains the IPC permissions stored at the same positions.

DTOS Kernel Definition 29

$$
\begin{array}{l}
Ipc\_permissions \\
\cap\ (Memory\_object\_permissions \cup Pager\_permissions \\
\quad \cup\ Thread\_permissions \cup Task\_permissions \\
\quad \cup\ Host\_name\_port\_permissions \cup Host\_control\_port\_permissions \\
\quad \cup\ Processor\_permissions \cup Procset\_permissions \\
\quad \cup\ Device\_permissions \cup Kernel\_reply\_permissions) \\
= \emptyset
\end{array}
$$

5.5 Access Vector Cache

The kernel receives an access decision from the security server as a $Ruling$. Each ruling consists of:

- ssi — a subject security identifier
- osi — an object security identifier
- $access\_vector$ — a set of granted permissions between the ssi and osi
- $control\_vector$ — the set of granted permissions which are allowed to be cached in the kernel for later access
- $expiration\_value$ — the time at which the cached permissions expire

DTOS Kernel Definition 30

$$
\begin{array}{l}
\textit{Ruling} \\
\hline
ssi : SSI \\
osi : OSI \\
access\_vector : \mathbb{P}\ PERMISSION \\
control\_vector : \mathbb{P}\ PERMISSION \\
expiration\_value : \mathbb{N}
\end{array}
$$

Review Note:
We need to be careful not to get bit by using ssi and osi in $Ruling$, since they are often used as "variables" also. Or else we could rename them here.

A ruling is usable for a given ssi and osi if the ssi and osi match those in the ruling and the ruling has not expired. The expression $Usable\_ruling(ssi, osi, time)$ denotes the set of all such rulings with respect to ssi, osi and $time$, the time at which the ruling is consulted. When a ruling is initially received by the kernel, the kernel need only check the access vector and expiration time to see if a permission is granted. This is reflected by the function $Ruling\_allows(ruling, ssi, osi)$ which returns the set of permissions in the access vector of $ruling$ if ssi and osi are the same as in $ruling$.

Editorial Note:
The prototype does not currently check the expiration time in these cases, but we plan to correct this.

DTOS Kernel Definition 31

$$
\begin{array}{l}
Usable\_ruling : SSI \times OSI \times \mathbb{N} \longrightarrow \mathbb{P}\ Ruling \\
Ruling\_allows : Ruling \times SSI \times OSI \times \mathbb{N} \longrightarrow \mathbb{P}\ PERMISSION \\
\hline
\forall\ ruling : Ruling;\ ssi : SSI;\ osi : OSI;\ time : \mathbb{N};\ permission : PERMISSION \\
\bullet\ (ruling \in Usable\_ruling(ssi, osi, time) \\
\quad \Leftrightarrow (ssi = ruling.ssi \\
\qquad \wedge\ osi = ruling.osi \\
\qquad \wedge\ time < ruling.expiration\_value)) \\
\wedge\ permission \in Ruling\_allows(ruling, ssi, osi, time) \\
\quad \Leftrightarrow (ruling \in Usable\_ruling(ssi, osi, time) \\
\qquad \wedge\ permission \in ruling.access\_vector)
\end{array}
$$

To enhance performance, the kernel is permitted to cache the rulings provided by security servers. A cached ruling is usable for a given ssi, osi and permission if the ssi and osi match those in the ruling, the permission is in the $control\_vector$ and the ruling has not expired. The expression $Usable\_cached\_ruling(ssi, osi, permission, time)$ denotes the set of all such rulings. Once cached, a ruling grants a particular $permission$ from ssi to osi if the ruling is usable and the permission is included in the $access\_vector$. This is reflected by the function $Cached\_ruling\_allows(ruling, ssi, osi, time)$, where $time$ is the time at which the ruling is consulted.

DTOS Kernel Definition 32

$$
\begin{array}{l}
Usable\_cached\_ruling : SSI \times OSI \times PERMISSION \times \mathbb{N} \longrightarrow \mathbb{P}\ Ruling \\
Cached\_ruling\_allows : Ruling \times SSI \times OSI \times \mathbb{N} \longrightarrow \mathbb{P}\ PERMISSION \\
\hline
\forall\ ruling : Ruling;\ ssi : SSI;\ osi : OSI;\ time : \mathbb{N};\ permission : PERMISSION \\
\bullet\ (ruling \in Usable\_cached\_ruling(ssi, osi, permission, time) \\
\quad \Leftrightarrow (ruling \in Usable\_ruling(ssi, osi, time) \\
\qquad \wedge\ permission \in ruling.control\_vector)) \\
\wedge\ (permission \in Cached\_ruling\_allows(ruling, ssi, osi, time) \\
\quad \Leftrightarrow (ruling \in Usable\_cached\_ruling(ssi, osi, permission, time) \\
\qquad \wedge\ permission \in ruling.access\_vector))
\end{array}
$$

The kernel cache is a set of rulings, represented by $\underline{c}ache$. There may only be one unexpired ruling in the cache for each (ssi, osi) pair. The function $cache\_allows(ssi, osi)$ returns the set of permissions granted to the (ssi, osi) pair by the rulings in the cache according to the function $Cached\_ruling\_allows$. The quadruple $(ssi, osi, permission, ruling)$ is in $cached\_ruling\_avail$ if and only if $ruling$ is in the cache and it is usable for ssi, osi and $permission$ at the current time.

DTOS Kernel Definition 33

$$
\begin{array}{l}
\textit{KernelCache} \\
\hline
\underline{c}ache : \mathbb{P}\ Ruling \\
cache\_allows : SSI \times OSI \longrightarrow \mathbb{P}\ PERMISSION \\
cached\_ruling\_avail : \mathbb{P}(SSI \times OSI \times PERMISSION \times Ruling) \\
HostTime \\
\hline
\forall\ ruling_1, ruling_2 : Ruling \\
\mid \{ ruling_1, ruling_2 \} \subseteq \underline{c}ache \\
\quad \wedge\ ruling_1.ssi = ruling_2.ssi \\
\quad \wedge\ ruling_1.osi = ruling_2.osi \\
\quad \wedge\ ruling_1.expiration\_value > \underline{h}ost\_time \\
\quad \wedge\ ruling_2.expiration\_value > \underline{h}ost\_time \\
\bullet\ ruling_1 = ruling_2 \\
\\
\forall\ ssi : SSI;\ osi : OSI \\
\bullet\ cache\_allows(ssi, osi) = \bigcup\{ruling : Ruling \mid ruling \in \underline{c}ache \\
\quad \bullet\ Cached\_ruling\_allows(ruling, ssi, osi, \underline{h}ost\_time)\} \\
\\
\forall\ ssi : SSI;\ osi : OSI;\ permission : PERMISSION;\ ruling : Ruling \\
\bullet\ (ssi, osi, permission, ruling) \in cached\_ruling\_avail \\
\quad \Leftrightarrow (ruling \in \underline{c}ache \\
\qquad \cap\ Usable\_cached\_ruling(ssi, osi, permission, \underline{h}ost\_time))
\end{array}
$$

5.6 Message Security Information

Each existing message has an SSI associated with it that indicates the SSI of the task that sent the message. The expression $\underline{m}sg\_sending\_sid(message)$ indicates the SSI of the task that sent $message$. In addition, certain messages have an associated SSI that indicates which tasks may receive the message. The set $msg\_receiver\_specified$ indicates the set of messages that have a receiving SID specified, and $\underline{m}sg\_receiving\_sid(message)$ indicates the receiving SSI for each message in this set. As part of the processing of a message, the sender's permissions to the destination port are computed and attached to the message. The set $msg\_ruling\_computed$ denotes the set of messages for which the permissions have already been computed, and $\underline{m}sg\_ruling(message)$ indicates the associated set of permissions for each such message. A ruling must be computed for each message before the message can be enqueued at a port. An "effective" sending SID and access vector may optionally be specified by the sender of a message. The expressions $\underline{m}sg\_specified\_sid(message)$ and $\underline{m}sg\_specified\_vector(message)$ indicate, respectively, the "effective" SID and access vector specified by the sender.

Editorial Note:
Need to think about how to model the specified vectors. The current specification ignores the cache control and notification vectors. The prototype currently has all three vectors represented explicitly. It has been implemented to allow the number of vectors to be easily changed.

DTOS Kernel Definition 34

$DtosMessages$

$MessageExist$
$MessageQueues$
$\underline{m}sg\_sending\_sid : MESSAGE \pfun SSI$
$msg\_receiver\_specified : \mathbb{P}\ MESSAGE$
$\underline{m}sg\_receiving\_sid : MESSAGE \pfun SSI$
$msg\_ruling\_computed : \mathbb{P}\ MESSAGE$
$\underline{m}sg\_ruling : MESSAGE \pfun Ruling$
$\underline{m}sg\_specified\_sid : MESSAGE \pfun SSI$
$\underline{m}sg\_specified\_vector : MESSAGE \pfun \mathbb{P}\ PERMISSION$

$\mathrm{dom}\ \underline{m}sg\_sending\_sid = \underline{m}essage\_exists$
$\mathrm{dom}\ \underline{m}sg\_receiving\_sid = msg\_receiver\_specified \subseteq \underline{m}essage\_exists$
$\mathrm{dom}\ \underline{m}sg\_ruling = msg\_ruling\_computed \subseteq \underline{m}essage\_exists$
$\mathrm{dom}\ containing\_port \subseteq msg\_ruling\_computed$
$\mathrm{dom}\ \underline{m}sg\_specified\_sid \subseteq \underline{m}essage\_exists$
$\mathrm{dom}\ \underline{m}sg\_specified\_vector \subseteq \underline{m}essage\_exists$

5.7 Task Creation Information

Each task has a state used in controlling the secure initiation of threads within that task. The type $TASK\_CREATION\_STATE$ is comprised of the possible values of this state. The recognized values of this type are:

- Tcs_task_empty — indicates a task that was created using **task_create_secure** and does not yet have any threads.
- $Tcs\_thread\_created$ — indicates a task created using **task_create_secure** for which a thread has been created using **thread_create_secure** but has not had its initial state set.

- $Tcs\_thread\_state\_set$ — indicates a task created using **task_create_secure** for which a thread has been created using **thread_create_secure** that has had its initial state set using **thread_set_state_secure** but has not been resumed (i.e., started).
- Tcs_task_ready —- indicates either a task that was not created using **task_create_secure** or a task that was created using **task_create_secure** and which has a thread that was created using **thread_create_secure**, has had its state set using **thread_set_state_secure**, and has been resumed using **thread_resume_secure**.

These states are used to ensure that processes initiated using **task_create_secure** follow the normal process initiation sequence of:

1. Create the task.
2. Create a thread within the task.
3. Set the state of the thread.
4. Resume the thread.

Review Note:
The above, particularly the description of Tcs_task_ready, must be checked against the prototype

---

This allows an untrusted process to create a trusted process using **task_create_secure** while prohibiting the untrusted process from (for example) changing the state of threads in the trusted process after the trusted process has started execution.

The expression $\underline{t}ask\_creation\_state(task)$ denotes the creation state of $task$.

DTOS Kernel Definition 35

$$TASK\_CREATION\_STATE ::= Tcs\_task\_empty \mid Tcs\_thread\_created$$
$$\mid Tcs\_thread\_state\_set \mid Tcs\_task\_ready$$

TaskCreationState

$$TaskExist$$
$$\underline{t}ask\_creation\_state : TASK \mathrel{+\!\!\!\rightarrow} TASK\_CREATION\_STATE$$

$$\mathrm{dom}\, \underline{t}ask\_creation\_state = \underline{t}ask\_exists$$

The Mach model of process creation uses an existing task to serve as a “template” for each new task. This task is the `parent_task` parameter to **task_create**. A newly created task inherits parts of its environment, such as portions of its address space, from the “parent” task. To simplify the statement of the security requirements on task creation, we introduce $\underline{p}arent\_task(task)$ to denote $task$'s parent.[10]

DTOS Kernel Definition 36

ParentTask

$$\underline{p}arent\_task : TASK \mathrel{+\!\!\!\rightarrow} TASK$$

[10] Note that this information is not actually recorded in the current design. Since we only use this information for stating requirements on task creation and this information is available at this point in the processing in the implementation, this deviation between the model and the implementation is tolerable.

5.8 Server Ports

The kernel records the ports to be used for communications with certain servers:

- $\underline{s}ecurity\_server\_master\_port$ denotes the port used by the kernel to make requests of the security server.
- $\underline{s}ecurity\_server\_client\_port$ denotes the port used by non-kernel clients to make requests of the security server.
- $\underline{a}uthentication\_server\_port$ denotes the port used to make requests of the authentication server.
- $\underline{a}udit\_server\_port$ denotes the port used to make requests of the audit server.
- $\underline{c}rypto\_server\_port$ denotes the port used to make requests of the crypto server.
- $\underline{n}egotiation\_server\_port$ denotes the port used to make requests of the negotiation server.
- $\underline{n}etwork\_ss\_port$ denotes the port used to make security requests over the network.

DTOS Kernel Definition 37

$ServerPorts$

$\underline{s}ecurity\_server\_master\_port : PORT$
$\underline{s}ecurity\_server\_client\_port : PORT$
$\underline{a}uthentication\_server\_port : PORT$
$\underline{a}udit\_server\_port : PORT$
$\underline{c}rypto\_server\_port : PORT$
$\underline{n}egotiation\_server\_port : PORT$
$\underline{n}etwork\_ss\_port : PORT$

When the kernel requests an access computation from the Security Server, it specifies a reply port to which the computed accesses should be sent. We use $\underline{k}ernel\_reply\_ports$ to denote the set of ports that the kernel has specified as reply ports for requests to the Security Server.

DTOS Kernel Definition 38

$KernelReplyPorts$

$PortExist$
$\underline{k}ernel\_reply\_ports : \mathbb{P}\ PORT$

$\underline{k}ernel\_reply\_ports \subseteq \underline{p}ort\_exists$

5.9 Memory Region Protections

The current protection of a region limits a task's access to that region. It is calculated as the intersection of the Mach protection together with the accesses allowed for a task to a memory region by the relevant access vector. We use $protection(task, index)$ to denote current protections of the region denoted by a given task-index pair.[11]

Mach Definition 108

[11] The prototype does not currently implement the enforcement of read-only access. The low-level memory routines in the prototype treat read and execute interchangeably.

$Protection$

$MachProtection$
$protection : (TASK \times PAGE\_INDEX) \rightarrow\!\!\!\!\!+\!\!\rightarrow \mathbb{P}\, PROTECTION$

$\mathrm{dom}\, protection = \mathrm{dom}\, \underline{m}ach\_protection$
$\forall\, task\_page\_index : TASK \times PAGE\_INDEX$
$\mid task\_page\_index \in \mathrm{dom}\, protection$
$\bullet\, protection(task\_page\_index) \subseteq \underline{m}ach\_protection(task\_page\_index)$

5.10 Summary of DTOS Kernel State

The DTOS kernel state is the Mach kernel state augmented with the access vector cache and the security information associated with subjects, objects, and messages.

DTOS Kernel Definition 39

$DtosAdditions$

$SubjectSid$
$ObjectSid$
$TargetSids$
$KernelAs$
$KernelCache$
$DtosMessages$
$TaskCreationState$
$ParentTask$
$ServerPorts$
$KernelReplyPorts$
$Protection$

DTOS Kernel Definition 40

$Dtos$

$Mach$
$DtosAdditions$

Section 6

Kernel Execution Model

This section describes the computational model used to represent the DTOS kernel requests and the additional data structures that are required to support this computational model. The organization of this section is as follows:

- Section 6.1, **Execution Summary**, gives a high level overview of the execution model and its data structures. The following sections give detailed descriptions of transitions which occur in the processing of all requests.
- Section 6.2, **Utility Transitions**, discusses several utility transitions that are used in various specifications.
- Section 6.3, **Trap Invocation**, discusses the transitions which occur at the start of any request.
- Sections 6.4 through 6.6 describe the initial processing common to all kernel requests which are made through the **mach_msg** trap.
- Section 6.7, **Definitions**, defines the data structures used to implement the transitions discussed in the previous sections.

6.1 Execution Summary

The DTOS execution model centers on the selection of a set of common "break points" in the processing of a kernel request. The break points are chosen to highlight significant processing events such as request invocation and service checking. We address the issue of atomicity by selecting an appropriate number and type of break points. One advantage of this approach is flexibility with respect to the level of detail in the model; we can easily change the amount of concurrency and level of detail by redefining the break points and the transitions which govern them.

Given a set of break points, every kernel request can be viewed as a sequence of transitions describing how processing moves from one break point to the next. In order to specify these transitions we need to know the current execution state for each thread. In the model, we maintain the execution status of each existing thread by setting the values of a function called $breaks$. The domain of this function is the set of existing threads and the values indicate what type of transitions have occurred as well as what information the thread needs to resume processing. In a sense the $breaks$ function is analogous to a processor's stack where information is stored between context switches, although the particular break points modeled by $breaks$ do not in general coincide with actual context switches.

Every thread executing in user space maintains a value of Bk_user_space. To enter kernel space a thread issues an instruction to trap into kernel code. We model such a transition by changing the relevant value of the function $breaks$ from Bk_user_space to Bk_new_trap. Similarly, we model a transition where a thread starts at a break point labeled Bk_point_A and ends at a break point labeled Bk_point_B by changing the relevant value of the function $breaks$ from Bk_point_A to Bk_point_B. The following sections describe the specific break points, their interpretations in the execution model, the information needed to resume processing, and the flow of processing from one break point to the next.

In the Mach kernel, many different requests share common features of processing. Another advantage of our model of execution is that it is easy to specify transitions which are common to many requests in a reusable manner. In general we begin by discussing transitions which are common to all requests and then discuss specialized transitions and finally discuss the transitions which are specific to a particular request. This section details the common transitions, transitions common to a class of requests are discussed in request chapter introductions, while the request specific transitions are specified in individual request sections.

Now we describe the data structures constituting the execution model. The values of $breaks$ are elements of the free type $BREAK\_STATUS$. These values indicate the current processing status of a thread together with the environment needed to resume processing. The elements of $BREAK\_STATUS$ are discussed in the following sections as preconditions and postconditions to transitions. The formal definition of $BREAK\_STATUS$ is given in Section 6.7.5. We use the schema $Breaks$ to define the $breaks$ function.

```
── Breaks ──────────────────────────────
│ ThreadExist
│ breaks : THREAD ⇸ BREAK_STATUS
├──────────
│ dom breaks = thread_exists
└────────────────────────────────────────
```

The state for the DTOS execution model consists of the components present in the DTOS kernel state together with the function $breaks$.

```
── DtosExec ────────────────────────────
│ Dtos
│ Breaks
└────────────────────────────────────────
```

We introduce a special schema $Transition$ which serves as the signature for every main transition. This schema introduces the DTOS kernel state and declares four variables. The first, $cpu??$, is the processor on which the transition is occurring. The other three are derived from $cpu??$ and are included as aliases to commonly used state elements. The variable $curr\_th??$ is the thread which is currently executing on $cpu??$, $curr\_task??$ is $curr\_th??$'s parent task, and $curr\_bk??$ is the execution status of $curr\_th??$.[12]

```
── Transition ──────────────────────────
│ Δ DtosExec
│ cpu?? : PROCESSOR
│ curr_th?? : THREAD
│ curr_task?? : TASK
│ curr_bk?? : BREAK_STATUS
├──────────
│ cpu?? ∈ dom active_thread
│ curr_th?? = active_thread(cpu??)
│ curr_task?? = owning_task(curr_th??)
│ curr_bk?? = breaks(curr_th??)
└────────────────────────────────────────
```

[12] The double questionmark decoration is used to provide a distinct look to these four components, since they have an interpretation distinct from either elements of the system state or inputs or outputs.

6.2 Utility Transitions

We begin our discussion of break points by specifying several transitions, shown in Figure 1, which are used as utilities. Each box in the diagram represents a complete transition; the first line in each box gives the name of the corresponding transition schema while the next two lines describe the break type of the precondition and postcondition, respectively. For example, the right-most box in Figure 1 describes the transition $RulingInCache$ which has as precondition the existence of a break of type $Bk\_check\_pending$, and produces a break of type Bk_have_ruling as a postcondition. In these diagrams, a solid arrow from *TransitionOne* to *TransitionTwo* indicates that *TransitionOne* precedes *TransitionTwo* **and** no other transitions from this request intervene (of course concurrency allows transitions from other requests to occur.) By contrast, a dashed line indicates that intervening requests may occur. For example, the line from $RulingNotInCache$ to $RulingObtained$ is dashed to reflect the fact that when the kernel waits for a ruling from the security server the most general interaction could involve repeated failures and retries.

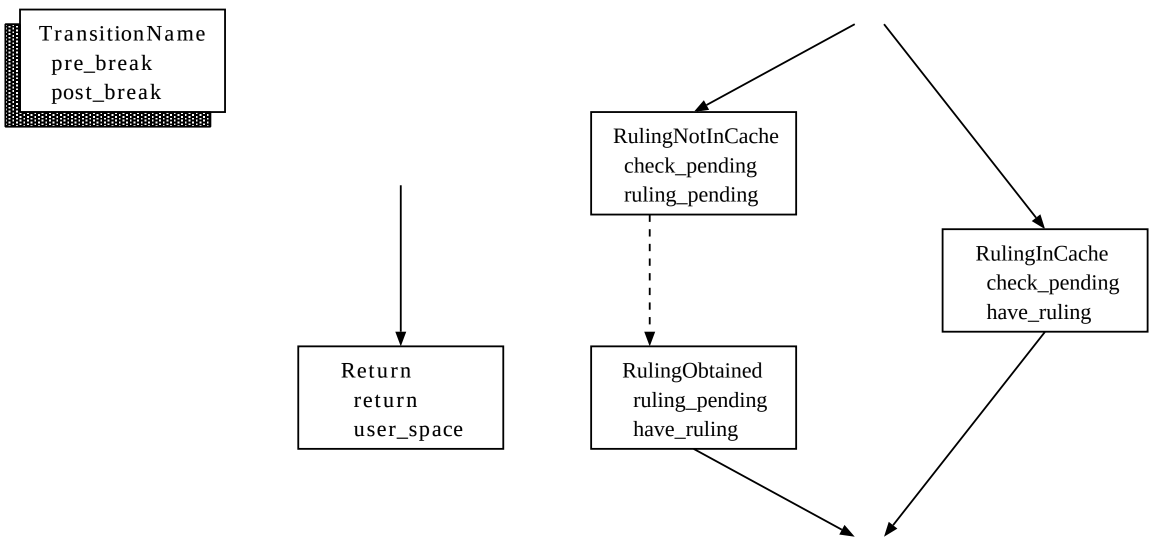

Figure 1: Utility Transitions

6.2.1 The Return Utility

Here we discuss the transition associated with request termination. The final transition in the processing of an IPC based request occurs when the kernel builds a return message containing status information and a specific kernel reply. We model such a transition with the schema $Return$. The precondition for this transition is the existence of a break of type Bk_return and the post condition is the creation of a break of type Bk_user_space signaling the fact that the thread has left kernel processing. The break Bk_user_space does not store any environment parameters (since no further processing is necessary) while Bk_return maintains the following information:

- $reply\_to\_port$ — the port where the return message should be enqueued (if dead or null no message is sent),
- $operation$ — the operation identifier for the terminating request,

- *reply* — an element of the type $KERNEL\_REPLY$ representing request specific output parameters supplied by the kernel,
- *return* — an element of the type $KERNEL\_RETURN$ describing the error status of request processing.

The kernel uses these parameters to build the return message. We model message construction with a set of functions: $Outputs\_to\_body$, $Reply\_size$, $Reply\_complex$, and $Reply\_op$. These functions and the types $KERNEL\_REPLY$ and $KERNEL\_RETURN$ are discussed in detail in Section 6.7.1. It is worth noting that this reply message is not sent via **mach_msg**; rather the kernel builds the message and directly enqueues it at $reply\_to\_port$.

Editorial Note:
We do not currently specify enqueueing of the message.

Return

Transition

$$
\begin{array}{l}
\exists\, message : MESSAGE;\ int\_msg : InternalMessage;\\
\quad reply\_to\_port : PORT;\ operation : OPERATION;\\
\quad reply : KERNEL\_REPLY;\ return : KERNEL\_RETURN\\
\bullet\ curr\_bk?? = Bk\_return(reply\_to\_port, operation, reply, return)\\
\quad \land\ message \notin \underline{m}essage\_exists\\
\quad \land\ \underline{m}essage\_exists' = \underline{m}essage\_exists \cup \{ message \}\\
\quad \land\ \underline{m}sg\_contents(message) = int\_msg\\
\quad \land\ int\_msg.header.local\_rights = \varnothing\\
\quad \land\ int\_msg.header.complex\\
\qquad = Reply\_complex(operation, Outputs\_to\_body(reply, return))\\
\quad \land\ int\_msg.header.size\\
\qquad = Reply\_size(operation, Outputs\_to\_body(reply, return))\\
\quad \land\ int\_msg.header.remote\_port = reply\_to\_port\\
\quad \land\ int\_msg.header.local\_port = \varnothing\\
\quad \land\ int\_msg.header.operation = Reply\_op(operation)\\
\quad \land\ int\_msg.body = Outputs\_to\_body(reply, return)\\
\quad \land\ int\_msg.option = \{ Mach\_send\_msg \}\\
\quad \land\ breaks' = breaks \oplus \{ curr\_th?? \mapsto Bk\_user\_space \}
\end{array}
$$

6.2.2 Permission Checking

Next we define the set of transitions involved in specifying a permission check. There are two possible transitions at the start of a permission check: $RulingInCache$ or $RulingNotInCache$. The precondition for each of these transitions is the existence of a break of type $Bk\_check\_pending$. The transition $RulingInCache$ examines the cache, determines that a cached ruling is available for the permission check, and creates a new break of type Bk_have_ruling. The transition $RulingNotInCache$ examines the cache, determines that a cached ruling is not available, and creates a new break of type $Bk\_ruling\_pending$. In this case the kernel continues processing by consulting the Security Server. We model this transition by the schema $RulingObtained$ which has as precondition the existence of a break of type $Bk\_ruling\_pending$ and which creates a new break of type Bk_have_ruling.

The permission checking transitions need to maintain several environment parameters. These are interpreted as:

- ssi — the subject SID of the check,
- osi — the object SID of the check,
- $perm$ — the required permission,
- env — the stored environment needed to resume processing,
- $op\_allowed$ — the boolean flag determining permission.

The first four of these are used in several places so we combine them in a structure called $CheckPending$:

$$\begin{array}{l} \textit{CheckPending} \\ \hline ssi : SSI \\ osi : OSI \\ perm : PERMISSION \\ env : ENVIRONMENT \\ \hline \end{array}$$

There are three distinct contexts in which a permission check may be required: at the beginning of a system trap (e.g. **mach_thread_self**), at the beginning of an IPC based request (e.g. the service check for **thread_get_state**), or later in the processing of an IPC based request (e.g. the deferred check in **thread_get_special_port**.) As such, the parameter env needs to store one of three different types of data. To handle these three cases we define a free type called $ENVIRONMENT$, which is described in Section 6.7.4.

The first permission checking utility transition is $RulingInCache$. When a permission check is initiated, the kernel consults the cache to determine if there is an applicable ruling. The schema $RulingInCache$ models the case where a permission check has been requested, and the kernel verifies that the cache contains an applicable ruling. The precondition of $RulingInCache$ is the existence of a break of type $Bk\_check\_pending$, reflecting the condition that the processing of some request is waiting for a permission check. The postcondition, Bk_have_ruling, reflects the fact that an available ruling was found; in this case the result of the permission check is stored in the parameter $perm$. The parameter env is passed along unchanged.

$$\begin{array}{l} \textit{RulingInCache} \\ \hline Transition \\ \hline \exists\, CheckPending;\ ruling : Ruling;\ op\_allowed : BOOLEAN \\ \bullet\ curr\_bk?? = Bk\_check\_pending(ssi, osi, perm, env) \\ \quad \wedge (ssi, osi, perm, ruling) \in cached\_ruling\_avail \\ \quad \wedge\ op\_allowed \\ \qquad = \mathbf{if}\ perm \in Cached\_ruling\_allows(ruling, ssi, osi, \underline{h}ost\_time) \\ \qquad \mathbf{then}\ True \\ \qquad \mathbf{else}\ False \\ \quad \wedge\ breaks' = breaks \oplus \{ curr\_th?? \mapsto Bk\_have\_ruling(perm, op\_allowed, env) \} \\ \hline \end{array}$$

The schema $RulingNotInCache$ models the case where a permission check has been requested and the kernel has determined that the cache does not contain an applicable ruling. Again the precondition is the existence of a break of type $Bk\_check\_pending$, but in this case the postcondition is a break of type $Bk\_ruling\_pending$ reflecting the fact that the kernel is waiting for a ruling from the Security Server. As before, the parameter env is passed along unchanged.

$$
\begin{array}{l}
\textit{RulingNotInCache} \\
\hline
\textit{Transition} \\
\hline
\exists\ \textit{CheckPending} \\
\bullet\ \textit{curr\_bk}?? = \textit{Bk\_check\_pending}(\textit{ssi}, \textit{osi}, \textit{perm}, \textit{env}) \\
\quad \wedge\ (\forall\ \textit{ruling} : \textit{Ruling} \mid \textit{ruling} \in \underline{c}\textit{ache} \\
\qquad \bullet\ \textit{ruling} \notin \textit{Usable\_cached\_ruling}(\textit{ssi}, \textit{osi}, \textit{perm}, \underline{h}\textit{ost\_time})) \\
\quad \wedge\ \textit{breaks}' = \textit{breaks} \oplus \{\ \textit{curr\_th}?? \mapsto \textit{Bk\_ruling\_pending}(\textit{ssi}, \textit{osi}, \textit{perm}, \textit{env})\ \}
\end{array}
$$

The schema *RulingObtained* models a transition where the kernel receives a valid ruling from the Security Server. The precondition is the existence of a break of type *Bk_have_ruling* and the postcondition is the creation of a new break of type *Bk_have_ruling*. The result of the permission check is stored in the parameter *perm*. As before, the parameter *env* is passed along unchanged.

Editorial Note:
The ruling obtained from the Security Server is modeled as a kernel input, but we do not specify how *ruling*? gets added to the cache.

$$
\begin{array}{l}
\textit{RulingObtained} \\
\hline
\textit{Transition} \\
\textit{ruling}? : \textit{Ruling} \\
\hline
\exists\ \textit{CheckPending};\ \textit{op\_allowed} : \textit{BOOLEAN} \\
\bullet\ \textit{curr\_bk}?? = \textit{Bk\_ruling\_pending}(\textit{ssi}, \textit{osi}, \textit{perm}, \textit{env}) \\
\quad \wedge\ \textit{ruling}? \in \textit{Usable\_ruling}(\textit{ssi}, \textit{osi}, \underline{h}\textit{ost\_time}) \\
\quad \wedge\ \textit{op\_allowed} \\
\qquad = \mathbf{if}\ \textit{perm} \in \textit{Ruling\_allows}(\textit{ruling}?, \textit{ssi}, \textit{osi}, \underline{h}\textit{ost\_time}) \\
\qquad \mathbf{then}\ \textit{True} \\
\qquad \mathbf{else}\ \textit{False} \\
\quad \wedge\ \textit{breaks}' = \textit{breaks} \oplus \{\ \textit{curr\_th}?? \mapsto \textit{Bk\_have\_ruling}(\textit{perm}, \textit{op\_allowed}, \textit{env})\ \}
\end{array}
$$

It is important to note that there are in fact many transitions between *RulingNotInCache* and *RulingObtained* including the sending of a message to the security server and receipt of a response.

6.3 Trap Invocation

All kernel requests are initiated through invocation of a trap while a thread is executing in user space. The thread must specify the particular trap identifier as well as some collection of parameters.

The precondition for this transition is the existence of a break of type *Bk_user_space* indicating that the thread is currently executing in user space. The postcondition is the creation of a new break of type *Bk_new_trap*. A break of type *Bk_new_trap* maintains two parameters: *trap_id*? identifies the type of trap being invoked and *user_spec*? contains components for the user supplied parameters. These types are described in Section 6.7.2 and Section 6.7.3.1.

$$
\begin{array}{l}
\textit{Invoke} \\
\hline
\textit{Transition} \\
trap\_id? : TRAP\_ID \\
user\_spec? : UserSpecified \\
\hline
curr\_bk?? = Bk\_user\_space \\
trap\_id? \in Trap\_ids \\
breaks' = breaks \oplus \{ curr\_th?? \mapsto Bk\_new\_trap(trap\_id?, user\_spec?) \}
\end{array}
$$

6.4 Initial **mach_msg** processing

In this section we discuss the transitions shown in Figure 2 which specify the early processing associated with the invocation of a **mach_msg** trap.

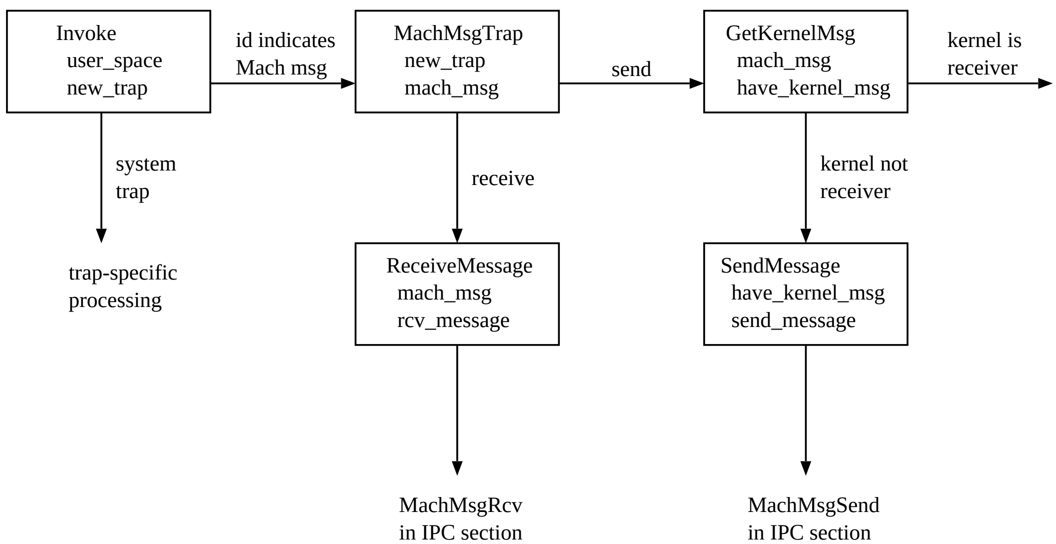

Figure 2: **mach_msg** Trap Invocation

If $trap\_id?$ indicates that the new break is $Mach\_msg\_trap$ processing continues with $MachMsgTrap$; the precondition is the existence of a break of type Bk_new_trap and the post-condition is the creation of a new break of type Bk_mach_msg. The break Bk_new_trap carries the trap identifier, $Mach\_msg\_trap$, as well as the user parameters $user\_spec$ which in this case contains the user space message.

$$
\begin{array}{l}
\textit{MachMsg} \\
\hline
\textit{Transition} \\
\hline
\exists\, user\_spec : UserSpecified \\
\bullet\ curr\_bk?? = Bk\_new\_trap(Mach\_msg\_trap, user\_spec) \\
\quad \wedge\ breaks' = breaks \oplus \{ curr\_th?? \mapsto Bk\_mach\_msg(user\_spec) \}
\end{array}
$$

Now we have two cases: this is a request to receive a message or a request to send a message.

Processing to receive a message is initiated in the transition $ReceiveMessage$. A client requests to receive a message by including $Mach\_rcv\_msg$ in the set of options. The precondition also includes the existence of a break of type Bk_mach_msg. The postcondition is the creation of a new break of type $Bk\_rcv\_message$. Subsequent processing of a receive request is described in Section C.1.

$$
\begin{array}{l}
\textit{ReceiveMessage} \\
\hline
\textit{Transition} \\
\hline
\exists\, user\_spec : UserSpecified \\
\bullet\ curr\_bk?? = Bk\_mach\_msg(user\_spec) \\
\quad \wedge\ Mach\_rcv\_msg \in user\_spec.options \\
\quad \wedge\ Mach\_send\_msg \notin user\_spec.options \\
\quad \wedge\ breaks' = breaks \oplus \{\, curr\_th?? \mapsto Bk\_rcv\_message(user\_spec) \,\}
\end{array}
$$

Otherwise this is a request to send a message, in which case processing continues with the conversion of the user space message into an internal representation. The transition $GetKernelMsg$ models a transition where the kernel resolves local name references to port references and virtual memory addresses to physical addresses. Here the precondition is the existence of a break of type Bk_mach_msg and the postcondition is a new break of type $Bk\_have\_kernel\_msg$. The kernel message is modeled by an element of type $InternalMessage$; this is discussed in Section 6.7.3.2.

Editorial Note:
Currently this is modeled as a “black box” transition, but the utilities exist (in the IPC section) to specify the conversion.

$$
\begin{array}{l}
\textit{GetKernelMsg} \\
\hline
\textit{Transition} \\
\hline
\exists\, user\_spec : UserSpecified;\ int\_msg : InternalMessage \\
\bullet\ curr\_bk?? = Bk\_mach\_msg(user\_spec) \\
\quad \wedge\ Mach\_send\_msg \in int\_msg.option \\
\quad \wedge\ breaks' = breaks \\
\qquad \oplus \{\, curr\_th?? \mapsto Bk\_have\_kernel\_msg(int\_msg) \,\}
\end{array}
$$

Now there are two cases to consider depending on whether or not the kernel is the receiver for the message.

If the kernel is not the receiver, this is a request to send a message and we model continued processing with the transition $SendMessage$. The precondition is the existence of a break of type $Bk\_have\_kernel\_msg$ and the postcondition is the creation of a new break of type $Bk\_send\_message$. Subsequent processing of a send request are described in Section C.1.

$$
\begin{array}{l}
\textit{SendMessage} \\
\hline
\textit{Transition} \\
\hline
\exists\, int\_msg : InternalMessage \\
\bullet\ curr\_bk?? = Bk\_have\_kernel\_msg(int\_msg) \\
\quad \wedge\ \underline{k}ernel \neq receiver(int\_msg.header.remote\_port) \\
\quad \wedge\ breaks' = breaks \oplus \{\, curr\_th?? \mapsto Bk\_send\_message(int\_msg) \,\}
\end{array}
$$

The case in which the kernel is the receiver is considered further in the next section.

6.5 Service Checks for IPC Based Kernel Requests

Kernel requests which are received through the **mach_msg** trap generally must pass through an initial service check to determine if the client has permission to make the request. This is performed whenever the permission required by the request is dependent only upon the client, the port provided as the “target” port in the request and the operation identifier. For a few requests, this information is not sufficient and the permission check is deferred.

Figure 3 shows the transitions described within this section.

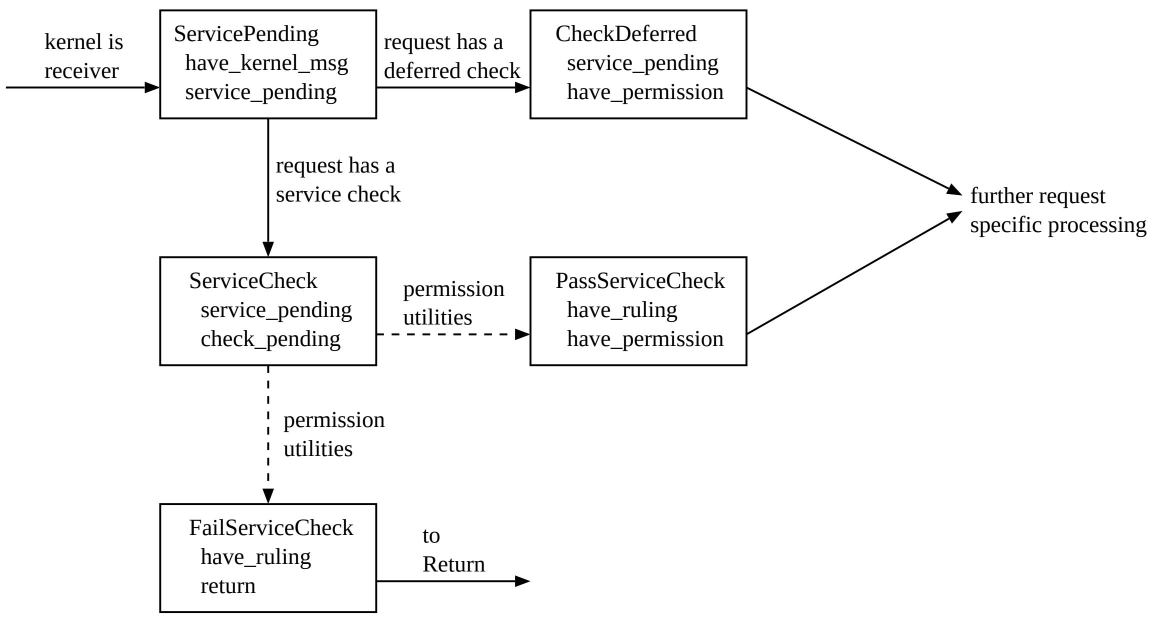

Figure 3: Message Transmission

The kernel prepares for the permission check by determining the operation that is being requested. We model this processing with the transition $ServicePending$. The precondition is the existence of a break of type $Bk\_have\_kernel\_msg$ and the postcondition is the creation of a new break of type $Bk\_service\_pending$. As before, the parameter int_msg is carried along for future use. In addition, this transition determines the operation and stores the value in the $operation$ parameter of $Bk\_service\_pending$.

$ServicePending$
$Transition$

$$\begin{array}{l}
\exists\, int_msg : InternalMessage \\
\bullet\ curr_bk?? = Bk_have_kernel_msg(int_msg) \\
\quad \wedge\ \underline{k}ernel = receiver(int_msg.header.remote_port) \\
\quad \wedge\ breaks' = breaks \\
\qquad \oplus \{\, curr_th?? \mapsto Bk_service_pending(int_msg, int_msg.header.operation)\,\}
\end{array}$$

Next the kernel determines whether or not the client has permission to request the operation. Each operation typically has an associated *primary permission* that a client must have in order to successfully call the operation. Checking that the client has this primary permission is referred to as making the *service check*. For example, the primary permission associated with the **thread_create** request is Add_thread. The **thread_create** request does not have any other permissions associated with it. As another example, the primary permission associated with the **mach_port_allocate** request is Add_name. However, there are other permissions such as $Hold\_receive$ that are also relevant to the **mach_port_allocate** request. The expression $Required\_permission(operation)$ denotes the primary permission, if any, associated with operation $operation$. For certain operations the service check is deferred because the required permission depends on a parameter that must first be processed. Once the parameter has been extracted from the message the appropriate permission check is performed. The set $Service\_check\_deferred$ is the set of all such operations. No operation in the domain of $Required\_permission$ can be in the set $Service\_check\_deferred$.

DTOS Kernel Definition 41

$$
\begin{array}{|l}
Required\_permission : OPERATION \pfun PERMISSION \\
Service\_check\_deferred : \mathbb{P}\ OPERATION \\
\hline
Service\_check\_deferred \cap \operatorname{dom} Required\_permission = \emptyset
\end{array}
$$

The schema $CheckDeferred$ models a transition in which the service check for a request is deferred until further processing can extract the appropriate parameters for the check. The precondition is the existence of a break of type $Bk\_service\_pending$ and the resulting break, $Bk\_have\_permission$, takes an element of type $PERMISSION$ as one of its arguments. We introduce a dummy permission called $permission\_deferred$ to act as the desired permission in a deferred service check.

$$
\begin{array}{|l}
permission\_deferred : PERMISSION
\end{array}
$$

$$
\begin{array}{|l}
\textit{CheckDeferred} \\
\hline
Transition \\
\hline
\exists\, int\_msg : InternalMessage;\ operation : OPERATION \\
\bullet\ curr\_bk?? = Bk\_service\_pending(int\_msg, operation) \\
\quad \land\ operation \in Service\_check\_deferred \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_have\_permission(int\_msg, permission\_deferred) \} \\
\hline
\end{array}
$$

If the service check is not deferred, the schema $ServiceCheck$ models a transition where the current thread waits for the kernel to obtain a ruling from the cache or from the security server. The precondition is the existence of a break of type $Bk\_service\_pending$ and the resulting break, $Bk\_check\_pending$, contains the parameters for the permission check. The break $Bk\_check\_pending$ requires an element of type $ENVIRONMENT$; the expression $E\_kern(int\_msg)$ packages the internal message into an element of this type. The function E_kern is discussed in Section 6.7.4.

$$
\begin{array}{l}
\textit{ServiceCheck} \\
\hline
\textit{Transition} \\
\hline
\exists\, int\_msg : InternalMessage;\ operation : OPERATION; \\
\quad ssi : SSI;\ osi : OSI;\ perm : PERMISSION \\
\bullet\ curr\_bk?? = Bk\_service\_pending(int\_msg, operation) \\
\quad \land\ operation \notin Service\_check\_deferred \\
\quad \land\ ssi = \underline{t}ask\_sid(curr\_task??) \\
\quad \land\ osi = \underline{p}ort\_sid(int\_msg.header.remote\_port) \\
\quad \land\ perm = Required\_permission(operation) \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, perm, E\_kern(int\_msg)) \}
\end{array}
$$

Editorial Note:
The OSI in the previous might need more elaboration depending upon the use of the “self” SIDs. Also, if we ever began to correctly consider specified sender SIDs, that would need to be taken care of around this point in the processing.

After obtaining a ruling, the kernel examines the access vector to determine if the operation is allowed. If the check fails, the kernel builds a return message by extracting information from the internal message. The $local\_port$ component of the message header specifies the reply port (Ip_null indicates that no reply should be sent.) In the case of a failed permission check we use the element $Null\_reply$ of type $KERNEL\_REPLY$ to represent an empty kernel reply and the kernel returns the special value $Kern\_insufficient\_permission$. We model this processing with the transition $FailServiceCheck$. The precondition is the existence of a break of type Bk_have_ruling indicating that the permission checking transition(s) have occurred. The postcondition is either Bk_return in the case where a return message has been requested or Bk_user_space for an immediate return to user space.

$$
\mid\ Null\_reply : KERNEL\_REPLY
$$

$$
\begin{array}{l}
\textit{FailServiceCheck} \\
\hline
\textit{Transition} \\
\hline
\exists\, perm : PERMISSION;\ op\_allowed : BOOLEAN;\ int\_msg : InternalMessage; \\
\quad reply\_to\_port : PORT;\ operation : OPERATION; \\
\quad reply : KERNEL\_REPLY;\ return : KERNEL\_RETURN \\
\bullet\ curr\_bk?? = Bk\_have\_ruling(perm, op\_allowed, E\_kern(int\_msg)) \\
\quad \land\ op\_allowed = False \\
\quad \land\ reply\_to\_port = int\_msg.header.local\_port \\
\quad \land\ operation = int\_msg.header.operation \\
\quad \land\ reply = Null\_reply \\
\quad \land\ return = Kern\_insufficient\_permission \\
\quad \land\ reply\_to\_port = Ip\_null \Rightarrow breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_user\_space \} \\
\quad \land\ reply\_to\_port \neq Ip\_null \Rightarrow breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_return(reply\_to\_port, operation, reply, return) \}
\end{array}
$$

Editorial Note:

The schema is not exactly coherent with the rest of the model. This is because the state model consider $local\_port$ to be a set of ports (zero or one element) rather than allowing it to take null values as it should and as is assumed here.

The transition $PassServiceCheck$ models the case were a ruling has been obtained and the operation is allowed. The precondition is the existence of a break of type Bk_have_ruling; such a break can only be produced be the permission checking utilities so this ensures that a permission check has occurred. The postcondition is the creation of a new break of type $Bk\_have\_permission$. The internal message is carried along as a parameter throughout the permission check as an element of type $ENVIRONMENT$ (in this case $E\_kern(int\_msg)$); in this transition we convert back to an element of type $InternalMessage$.

$PassServiceCheck$
$Transition$

$$\exists\, perm : PERMISSION;\ op\_allowed : BOOLEAN;\ int\_msg : InternalMessage$$
$$\bullet\ curr\_bk?? = Bk\_have\_ruling(perm, op\_allowed, E\_kern(int\_msg))$$
$$\quad \land\ op\_allowed = True$$
$$\quad \land\ breaks' = breaks$$
$$\qquad \oplus \{\, curr\_th?? \mapsto Bk\_have\_permission(int\_msg, perm)\,\}$$

In summary, if the service check passes or has been deferred there will be a break of type $Bk\_have\_permission$. Further processing is described in the next section.

6.6 Request Validation

The final request processing steps which are generally common to all IPC based kernel requests is validation of the request and extraction of the request parameters from the message body. These transitions are shown in Figure 4.

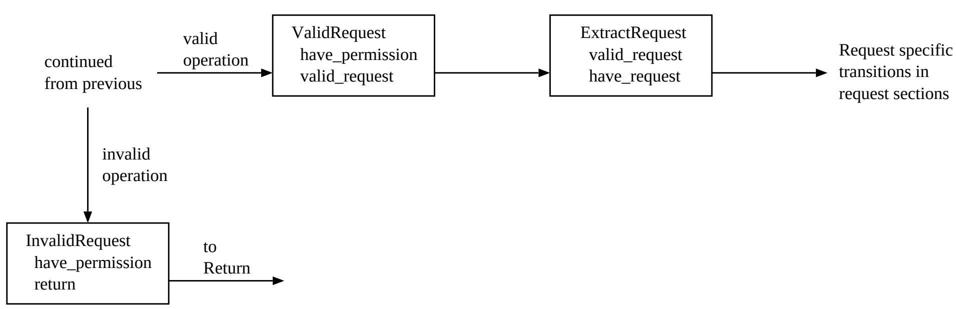

Figure 4: Request Validation

If the service check passes or is deferred the kernel next verifies that the specified operation is an allowed Mach operation. The set $Allowed\_kernel\_ops$ denotes the set of recognized Mach operations.

$$Allowed\_kernel\_ops : \mathbb{P}\ OPERATION$$

If the kernel determines that $operation$ is not an allowed kernel operation, an error message is generated and sent to the reply port. Again the kernel sends an empty reply, and if the reply port is null no message is sent. The kernel reply is the special value MIG_BAD_ID. This is modeled by the transition $InvalidRequest$. The precondition is a break of type $Bk\_have\_permission$ and the two possible postconditions are Bk_user_space for a return without message or Bk_return for a return with message.

$$MIG\_BAD\_ID : KERNEL\_REPLY$$

$InvalidRequest$

$Transition$

$$\begin{array}{l}
\exists\, int_msg : InternalMessage;\ perm : PERMISSION; \\
\quad reply_to_port : PORT;\ operation : OPERATION; \\
\quad reply : KERNEL_REPLY;\ return : KERNEL_RETURN \\
\bullet\ curr_bk?? = Bk_have_permission(int_msg, perm) \\
\quad \land\ operation = int_msg.header.operation \\
\quad \land\ operation \notin Allowed_kernel_ops \\
\quad \land\ reply_to_port = int_msg.header.local_port \\
\quad \land\ reply = MIG_BAD_ID \\
\quad \land\ return = Kern_invalid_value \\
\quad \land\ reply_to_port = Ip_null \Rightarrow breaks' = breaks \\
\qquad \oplus \{\, curr_th?? \mapsto Bk_user_space \,\} \\
\quad \land\ reply_to_port \neq Ip_null \Rightarrow breaks' = breaks \\
\qquad \oplus \{\, curr_th?? \mapsto Bk_return(reply_to_port, operation, reply, return) \,\}
\end{array}$$

Editorial Note:
The schema is not exactly coherent with the rest of the model. This is because the state model consider $local\_port$ to be a set of ports (zero or one element) rather than allowing it to take null values as it should and as is assumed here.

If the request is valid, we model processing with $ValidRequest$. The precondition is the existence of a break of type $Bk\_have\_permission$ and the postcondition $Bk\_valid\_request$ indicates that the operation is allowed.

$ValidRequest$

$Transition$

$$\begin{array}{l}
\exists\, int_msg : InternalMessage;\ perm : PERMISSION \\
\bullet\ curr_bk?? = Bk_have_permission(int_msg, perm) \\
\quad \land\ int_msg.header.operation \in Allowed_kernel_ops \\
\quad \land\ breaks' = breaks \oplus \{\, curr_th?? \mapsto Bk_valid_request(int_msg) \,\}
\end{array}$$

Finally, if the operation is valid, the kernel extracts the request parameters. We model this with the transition $ExtractRequest$. The precondition is the existence of a break of type $Bk\_valid\_request$ and the postcondition $Bk\_have\_request$, which maintains an element of type $Request$, indicates that a request has been extracted. The componenets of $Request$ are discussed in Section 6.7.3.3.

Editorial Note:
This is modeled as a “black box” conversion. Potentially there are many different extraction transitions, depending on the types of the parameters. At some point it is also important to deal with the possibility that the extraction code (MIG) gets too confused about the types.

ExtractRequest
Transition

$$\exists\, int\_msg : InternalMessage;\ request : Request$$
$$\bullet\ curr\_bk?? = Bk\_valid\_request(int\_msg)$$
$$\quad \wedge\ breaks' = breaks \oplus \{\, curr\_th?? \mapsto Bk\_have\_request(request)\,\}$$

Further request processing is discussed in the chapter introductions and request sections.

6.7 Definitions

In this section we define the types and constructors used to describe the break points.

6.7.1 Reply Messages

First we discuss the functions and types connected with kernel reply messages. We have two types to represent the information returned by the kernel in reply messages. The elements of $KERNEL\_REPLY$ represent the various types of output that the kernel can supply to a client through a reply message. Elements of type $KERNEL\_REPLY$ are request dependent, so here we define $KERNEL\_REPLY$ as an abstract set; particular elements are discussed in the request specifications.

$$[KERNEL\_REPLY]$$

The set $KERNEL\_RETURN$ is an enumerated type representing the possible return statuses that a request can generate. The set of statuses in DTOS consists of:

- $Kern\_success$ — the request was successful,
- $Kern\_failure$ — an implementation dependent failure occurred,
- $Kern\_invalid\_argument$ — an attempt was made to perform an operation on the wrong type of entity; for example, an attempt was made to perform a task operation on a thread,
- $Kern\_protection\_failure$ — an attempt was made to access memory in violation of the protections in force,
- $Kern\_invalid\_address$ — an invalid address was specified,
- $Kern\_no\_space$ — an attempt was made to allocate space in a task whose address space or name space was full,
- $Kern\_invalid\_host$ — an attempt was made to perform a host operation on an entity other than a host,
- $Kern\_resource\_shortage$ — insufficient resources were available for service to be provided,
- $Kern\_invalid\_right$ — the wrong type of port right was provided,
- $Kern\_invalid\_value$ — a parameter value that was out of range was provided,
- $Kern\_name\_exists$ — an attempt was made to reuse a name that was already used in the target task's address space,
- $Kern\_invalid\_name$ — a name provided as a port right was not currently in use,
- $Kern\_not\_in\_set$ — a name provided as the element of a port set was not in any port set,
- $Kern\_urefs\_overflow$ — an operation was attempted that would cause a user reference count to overflow,
- $Kern\_memory\_present$ —

Review Note:
Need to determine what this is used for if it is really used

- $Kern\_invalid\_task$ — an attempt was made to perform a task operation on an entity other than a task,
- $Kern\_eml\_bad\_cnt$ — an invalid syscall number was specified for an emulation vector entry,
- $Kern\_invalid\_capability$ — a provided name is not a right of the appropriate type,
- $Kern\_insufficient\_permission$ — a security checked failed in the processing of the request.

$[KERNEL\_RETURN]$

$$
\begin{array}{l}
Kern\_success : KERNEL\_RETURN \\
Kern\_failure : KERNEL\_RETURN \\
Kern\_invalid\_argument : KERNEL\_RETURN \\
Kern\_protection\_failure : KERNEL\_RETURN \\
Kern\_invalid\_address : KERNEL\_RETURN \\
Kern\_no\_space : KERNEL\_RETURN \\
Kern\_invalid\_host : KERNEL\_RETURN \\
Kern\_resource\_shortage : KERNEL\_RETURN \\
Kern\_invalid\_right : KERNEL\_RETURN \\
Kern\_invalid\_value : KERNEL\_RETURN \\
Kern\_name\_exists : KERNEL\_RETURN \\
Kern\_invalid\_name : KERNEL\_RETURN \\
Kern\_not\_in\_set : KERNEL\_RETURN \\
Kern\_urefs\_overflow : KERNEL\_RETURN \\
Kern\_true : KERNEL\_RETURN \\
Kern\_false : KERNEL\_RETURN \\
Kern\_memory\_present : KERNEL\_RETURN \\
Kern\_invalid\_task : KERNEL\_RETURN \\
Kern\_insufficient\_permission : KERNEL\_RETURN \\
Kern\_eml\_bad\_cnt : KERNEL\_RETURN \\
Kern\_invalid\_capability : KERNEL\_RETURN \\
\hline
Values\_disjoint\langle Kern\_success, Kern\_failure, Kern\_invalid\_argument, \\
\quad Kern\_protection\_failure, Kern\_invalid\_address, Kern\_no\_space, Kern\_invalid\_host, \\
\quad Kern\_resource\_shortage, Kern\_invalid\_right, Kern\_invalid\_value, Kern\_name\_exists, \\
\quad Kern\_invalid\_name, Kern\_not\_in\_set, Kern\_urefs\_overflow, \\
\quad Kern\_true, Kern\_false, Kern\_memory\_present, \\
\quad Kern\_eml\_bad\_cnt, Kern\_invalid\_task, \\
\quad Kern\_insufficient\_permission, Kern\_invalid\_capability\rangle
\end{array}
$$

Note that all but $Kern\_insufficient\_permission$ are Mach status codes, while $Kern\_insufficient\_permission$ is a DTOS addition.

Next we define several functions which package the return parameters into a message. The expression $Outputs\_to\_body(reply, return)$ converts a group of output parameters $reply$ and a status $return$ to the message body structure, and the expression $Reply\_size(operation, Outputs\_to\_body(reply, return))$ denotes the size of a message carrying $reply$ and $return$ as output from a request of type $operation$.

$$Outputs\_to\_body : KERNEL\_REPLY \times KERNEL\_RETURN \longrightarrow INTERNAL\_BODY$$
$$Reply\_size : OPERATION \times INTERNAL\_BODY \longrightarrow \mathbb{N}$$

As with general messages in Mach, a reply message can be either simple or complex as specified by the *complex* field of the message header. The expression $Reply\_complex(operation, Outputs\_to\_body(reply, return))$ denotes the value that should be assigned to this field when returning *reply* and *return* as output from a request of type *operation*.

$$Reply\_complex : OPERATION \times INTERNAL\_BODY \longrightarrow \mathbb{P}\ COMPLEX\_OPTION$$

The kernel also needs to assign a value to the *operation* field of the reply message. The expression $Reply\_op(operation)$ denotes the value that is used to indicate a reply message for a request of type *operation*.[13]

$$Reply\_op : OPERATION \rightarrowtail OPERATION$$

6.7.2 Trap Identifiers

We define the set $TRAP\_ID$ which represents the set of all trap operations. The set of identifiers used to represent traps is $Trap\_ids$.

$$[TRAP\_ID]$$

$$Evc\_wait\_trap, Mach\_thread\_self\_trap, Swtch\_trap, Swtch\_pri\_trap,$$
$$\quad Thread\_switch\_trap, Mach\_msg\_trap : TRAP\_ID$$
$$Trap\_ids : \mathbb{P}\ TRAP\_ID$$
$$\langle Evc\_wait\_trap, Mach\_thread\_self\_trap, Swtch\_trap, Swtch\_pri\_trap,$$
$$\quad Thread\_switch\_trap, Mach\_msg\_trap\rangle\ Values\_partition\ Trap\_ids$$

6.7.3 Environment Parameters

As processing of a request progresses, the parameters involved are subject to several transformations. To handle various data contexts we model three types of parameters: user specified parameters, kernel parameters, and abstract request parameters. In this section we define structures which represent these types of data.

As an example, there are three distinct contexts in which a permission check may be required: at the beginning of a system trap (e.g. **mach_thread_self**), at the beginning of an IPC based request (e.g. the service check for **thread_get_state**), or later in the processing of an IPC based request (e.g. the deferred check in **thread_get_special_port**.) As such, the parameter *env* supplied to the permission checking utilities needs to store one of three different types of data. To handle these three cases we define a free type called $ENVIRONMENT$, which uses three different constructor functions to store the three types of parameters.

6.7.3.1 User Parameters In the case of a system trap, we use the structure *UserSpecified*. The components consist of all possible user inputs for the various traps. Most of these inputs come from $Mach\_msg\_trap$. The components have the following interpretations:

[13] The current implementation defines $Reply\_op(operation)$ to be $operation + 100$.

- $trap\_id$ — the identifier of the originating trap,
- $priority$ — the priority argument to the **swtch_pri** trap,
- $thread\_switch\_name$ — the name argument to the **thread_switch** trap,
- $thread\_switch\_option$ — the option to the **thread_switch** trap,
- $timeout$ — a timeout parameter, used by the **thread_switch** and **mach_msg** traps,
- $message$ — the user message being sent via **mach_msg**,
- $options$ — the send/receive options specified in the **mach_msg** trap,
- $send\_size$ — the size of a message being sent,
- $receive\_size$ — the maximum size message that can be received,
- $receiver$ — where return messages will be received,
- $notify$ — where to send notifications.

The type $THREAD\_SWITCH\_OPTION$ consists of the following three values:

$$THREAD\_SWITCH\_OPTION ::= Thread\_switch\_none \mid Thread\_switch\_depress \\ \mid Thread\_switch\_wait$$

UserSpecified

$$\begin{array}{l} trap\_id : TRAP\_ID \\ priority : \mathbb{Z} \\ thread\_switch\_name : NAME \\ thread\_switch\_option : THREAD\_SWITCH\_OPTION \\ timeout : \mathbb{N} \\ message : Message \\ options : \mathbb{P}\ MACH\_MSG\_OPTION \\ send\_size : \mathbb{N} \\ receive\_size : \mathbb{N} \\ receiver : NAME \\ notify : NAME \end{array}$$

6.7.3.2 Kernel Parameters

In the case of a service check, the user space parameters contained in an IPC message have been converted into internal representations—names have become ports and virtual memory references have become physical addresses. We model this by storing the relevant processing information in a structure of type $InternalMessage$, which is described in Section 4.10.8.

6.7.3.3 Request Parameters

In the case of a deferred check, the kernel has performed additional processing on the message parameters to extract the request parameters. We use an element of type $Request$ to represent the parameters of the abstract service being requested. These values are obtained from the contents of the message and the port through which the message was received. The components have the following interpretations:

- $operation$ — the type of operation specified in the message,
- $service\_port$ — the port through which the message was received,
- pc — $service\_port$'s class (task, thread, memory control, ...),
- $reply\_to\_port$ — a set containing the port to which the reply message should be sent, the empty set indicates no reply should be sent,
- $message$ — the received message,
- $sending\_sid$ — the SID of the subject that sent the message ,

- *receiver_specified* — a Boolean indicating whether an intended receiving SID is specified,
- *receiving_sid* — the intended receiving SID (if any).

$BOOLEAN ::= True \mid False$

Request

$operation : OPERATION$
$service\_port : PORT$
$pc : PORT\_CLASS$
$reply\_to\_port : \mathbb{P}\ PORT$
$message : MESSAGE$
$sending\_sid : SSI$
$receiver\_specified : BOOLEAN$
$receiving\_sid : SSI$

6.7.4 Environment

The three preceding data types are used to build the free type $ENVIRONMENT$, which is used to store parameters between transition breaks:

$$
\begin{aligned}
ENVIRONMENT ::= {} & E\_user \langle\!\langle UserSpecified \rangle\!\rangle \\
& \mid E\_kern \langle\!\langle InternalMessage \rangle\!\rangle \\
& \mid E\_req \langle\!\langle Request \rangle\!\rangle
\end{aligned}
$$

6.7.5 Break Status

Finally, we give the formal definition of the free type $BREAK\_STATUS$. All of the constructor functions defining $BREAK\_STATUS$ have been discussed in the preceding sections. They consist of:

- *Bk_return* indicates that a return message is being built and processing is terminating (Section 6.2),
- *Bk_check_pending* indicates that a thread is waiting to check whether a usable ruling for a given permission is available in the cache (Section 6.2),
- *Bk_ruling_pending* indicates that a thread is waiting for a ruling to be received from the Security Server (Section 6.2),
- *Bk_have_ruling* indicates that a thread has obtained a ruling and permission (Section 6.2),
- *Bk_user_space* indicates that a thread is executing in user space (Section 6.3),
- *Bk_new_trap* indicates that a thread has issued a trap into kernel space (Section 6.3),
- *Bk_mach_msg* indicates that a thread has issued a trap of type *Mach_msg_trap* (Section 6.4),
- *Bk_rcv_message* indicates that a thread is waiting to receive a message (Section 6.4),
- *Bk_have_kernel_msg* indicates that a user space message has been converted to an internal kernel message (Section 6.4),
- *Bk_send_message* indicates that a thread is waiting to send a message and the kernel is not the receiver (Section 6.4),
- *Bk_service_pending* indicates that a thread has sent a message to the kernel and is waiting for an IPC based request to be performed (Section 6.5),

- $Bk\_have\_permission$ indicates that a permission check has terminated and the operation is allowed (Section 6.5),
- $Bk\_valid\_request$ indicates that the indicated operation is a recognized Mach operation (Section 6.6),
- $Bk\_have\_request$ indicates that the MIG interface has extracted the internal request parameters (Section 6.6).

The following abbreviations are used for some of the parameters in the constructor functions. Values of type $BK\_PERM\_REQUEST$ indicate the permission check for which a given request is waiting. Values of type BK_PERM_RESULT indicate the permission check and its results for a given request. Values of type BK_RETURN contain the information necessary to build a return message.

$$
\begin{array}{l}
BK\_PERM\_REQUEST == SSI \times OSI \times PERMISSION \times ENVIRONMENT \\
BK\_PERM\_RESULT == PERMISSION \times BOOLEAN \times ENVIRONMENT \\
BK\_RETURN == PORT \times OPERATION \times KERNEL\_REPLY \times KERNEL\_RETURN
\end{array}
$$

The type $BREAK\_STATUS$ is defined by:

$$
\begin{array}{l}
BREAK\_STATUS ::= Bk\_return\langle\langle BK\_RETURN\rangle\rangle \\
\quad | \ Bk\_check\_pending\langle\langle BK\_PERM\_REQUEST\rangle\rangle \\
\quad | \ Bk\_ruling\_pending\langle\langle BK\_PERM\_REQUEST\rangle\rangle \\
\quad | \ Bk\_have\_ruling\langle\langle BK\_PERM\_RESULT\rangle\rangle \\
\quad | \ Bk\_user\_space \\
\quad | \ Bk\_new\_trap\langle\langle Trap\_ids \times UserSpecified\rangle\rangle \\
\quad | \ Bk\_mach\_msg\langle\langle UserSpecified\rangle\rangle \\
\quad | \ Bk\_rcv\_message\langle\langle UserSpecified\rangle\rangle \\
\quad | \ Bk\_have\_kernel\_msg\langle\langle InternalMessage\rangle\rangle \\
\quad | \ Bk\_send\_message\langle\langle InternalMessage\rangle\rangle \\
\quad | \ Bk\_service\_pending\langle\langle InternalMessage \times OPERATION\rangle\rangle \\
\quad | \ Bk\_have\_permission\langle\langle InternalMessage \times PERMISSION\rangle\rangle \\
\quad | \ Bk\_valid\_request\langle\langle InternalMessage\rangle\rangle \\
\quad | \ Bk\_have\_request\langle\langle Request\rangle\rangle
\end{array}
$$

Section 7
System Trap Requests

7.1 Introduction to System Trap Requests

This chapter describes the system trap requests in DTOS.

7.1.1 Kernel Processing

Every trap begins with the transition $Invoke$ which produces a new break of type $Bk\_new\_trap(trap\_id?, user\_spec?)$. Typically there are no user inputs, but several traps (e.g. **thread_switch**) require one or more inputs. These parameters are stored in components of the $user\_spec?$ structure. Permission checks are handled by the utilities discussed in Section 6.2.

When a trap produces an error, it does not affect the system state. We define the following transition to describe the situation where trap processing encounters an error and execution leaves kernel space without changing the state.

$$\begin{array}{l}
\textit{TrapOnlyObserves} \\
\hline
\Xi\ Dtos \\
Transition \\
\hline
breaks' = breaks \oplus \{\ curr_th?? \mapsto Bk_user_space\ \}
\end{array}$$

We now describe the individual system trap requests.

7.1.2 Adding a Send Right

Editorial Note:
The following schema is used both in this chapter and the thread requests chapter. As currently used, it forces us to violate the goal of having a common signature for all state transition schemas. It is not clear that there is a simple way to avoid this, or whether this indicates a weakness in Z or in our use of Z.

The following schema describes a successful addition of a send right with name equal to $name$ for the port $port$ to the port name space for task $task$. This action is one result of several thread requests.

The name for the new right cannot already name a send-once right, dead name or port set. If it does name either a send or receive right, this right must be for $port$. If it is does not already name a send right, then a send right for port $port$ with name $name$ is added to the port name space for $task$ with a user reference count of 1. If it already names a send right belonging to $task$, the user reference count for this send right is incremented by 1.

$$
\begin{array}{l}
\textit{AddSendRight} \\
\hline
\Delta\ PortNameSpace \\
\Delta\ SendRightsCount \\
name : NAME \\
task : TASK \\
port : PORT \\
\hline
(task, name) \notin so\_right \cup dead\_namep \cup port\_set\_namep \\
(task, name) \in s\_r\_right \Rightarrow named\_port(task, name) = port \\
(task, name) \notin s\_right \\
\quad \Rightarrow \underline{p}ort\_right\_rel' = \underline{p}ort\_right\_rel \cup \{(task, port, name, Send, 1)\} \\
(task, name) \in s\_right \\
\quad \Rightarrow \underline{p}ort\_right\_rel' = \underline{p}ort\_right\_rel \\
\qquad \setminus \{(task, port, name, Send, s\_right\_ref\_count(task, name))\} \\
\qquad \cup \{(task, port, name, Send, s\_right\_ref\_count(task, name) + 1)\}
\end{array}
$$

7.2 mach_thread_self

The request **mach_thread_self** places a send right for a thread's kernel port in the name space of the owning task of the thread. It is a system trap.

7.2.1 Kernel Interface

```
mach_port_t mach_thread_self
        ();
```

7.2.1.1 Input Parameters

No input parameters are provided to the kernel for a **mach_thread_self** request.

7.2.1.2 Output Parameters

The following output parameters are returned by the kernel for a **mach_thread_self** request:

- $kernel\_port\_name!$ — the name for a send right to the thread's kernel port in the port name space of the thread's owning task

$$
\begin{array}{l}
\textit{MachThreadSelf Outputs} \\
\hline
kernel\_port\_name! : NAME
\end{array}
$$

7.2.2 Request Criteria

The following criteria are defined for the **mach_thread_self** request.

- **C1 —** Permission $Get\_thread\_kernel\_port$ has been obtained.

$C1MachThreadSelfGetThreadKernelPort$
$Transition$
$\exists\, env : ENVIRONMENT$
$\bullet\; curr\_bk?? = Bk\_have\_ruling(Get\_thread\_kernel\_port, True, env)$

$NotC1MachThreadSelfGetThreadKernelPort$
$Transition$
$\exists\, env : ENVIRONMENT$
$\bullet\; curr\_bk?? = Bk\_have\_ruling(Get\_thread\_kernel\_port, False, env)$

- **C2 —** The sself port for the current thread is not Ip_dead.

$C2MachThreadSelfKernelPortNotDead$
$Transition$
$thread\_sself(curr\_th??) \neq Ip\_dead$

$NotC2MachThreadSelfKernelPortNotDead$
$\quad \widehat{=}\; Transition \wedge \neg\, C2MachThreadSelfKernelPortNotDead$

- **C3 —** The sself port for the current thread is not Ip_null.

$C3MachThreadSelfKernelPortNotNull$
$Transition$
$thread\_sself(curr\_th??) \neq Ip\_null$

$NotC3MachThreadSelfKernelPortNotNull$
$\quad \widehat{=}\; Transition \wedge \neg\, C3MachThreadSelfKernelPortNotNull$

- **C4 —** Either the current task already holds a send or receive right to the thread's sself port (so that a new IPC entry need not be created), or the kernel has the available resources to create an IPC entry in the current task's name space. We do not actually model the consumption of resources by the kernel. So, we will use the set $Resources\_available\_to\_create\_ipc\_entry$ to indicate the set of states where there are sufficient resources to create an IPC entry.

$Resources\_available\_to\_create\_ipc\_entry : \mathbb{P}\, DtosExec$

$C4MachThreadSelfResourcesAvailable$
$Transition$
$((\exists\, name : NAME;\; i : \mathbb{N}_1;\; right : \{Send, Receive\}$
$\bullet\; (curr\_task??, thread\_sself(curr\_th??), name, right, i)$
$\qquad \in \underline{p}ort\_right\_rel)$
$\quad \vee\; \theta DtosExec \in Resources\_available\_to\_create\_ipc\_entry)$

$NotC4MachThreadSelfResourcesAvailable$
$\quad \widehat{=}\; Transition \wedge \neg\, C4MachThreadSelfResourcesAvailable$

- **C5 —** Permission $Hold\_send$ has been obtained.

$$
\begin{array}{|l}
\hline
C5MachThreadSelfHoldSend \\
Transition \\
\hline
\exists\, env : ENVIRONMENT \\
\bullet\ curr\_bk?? = Bk\_have\_ruling(Hold\_send, True, env) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
NotC5MachThreadSelfHoldSend \\
Transition \\
\hline
\exists\, env : ENVIRONMENT \\
\bullet\ curr\_bk?? = Bk\_have\_ruling(Hold\_send, False, env) \\
\hline
\end{array}
$$

7.2.3 Return Values

Table 1 describes the values returned at the completion of the request and the conditions under which each value is returned. We note that **C2** and **C3** may not both be false simultaneously.

$kernel\_port\_name!$	C1	C2	C3	C4	C5
$name$	T	T	T	T	T
$Mach\_port\_null$	T	T	T	T	F
$Mach\_port\_null$	T	T	T	F	-
$Mach\_port\_null$	T	T	F	-	-
$Mach\_port\_dead$	T	F	T	-	-
$Return\_status\_to\_name(Kern\_insufficient\_permission)$	F	-	-	-	-

Table 1: Return Values for **mach_thread_self**

We call attention here to the fact that **C1** – **C4** are checked in the the transition $MachThreadSelfMiddle$ (see below) while only **C5** is checked in transition $MachThreadSelfEnd$. These are distinct transitions between which an arbitrary number of other transitions may occur. Furthermore, **C5** is checked only if in some earlier transition for that trap request **C1** – **C4** are all true. We also note that $name$ is constrained by $AddSendRight$ when $RVMachThreadSelfGood$ is combined with $MachThreadSelfState$.

$$
\begin{array}{|l}
\hline
RVMachThreadSelfGood \\
C5MachThreadSelfHoldSend \\
MachThreadSelf\ Outputs \\
Transition \\
name : NAME \\
\hline
kernel\_port\_name! = name \\
breaks' = breaks \oplus \{ curr\_th?? \mapsto Bk\_user\_space \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
RVMachThreadSelfNoHoldSend \\
NotC5MachThreadSelfHoldSend \\
MachThreadSelf\ Outputs \\
\hline
kernel\_port\_name! = Mach\_port\_null \\
\hline
\end{array}
$$

$RVMachThreadSelfResourceShortage$

$C1MachThreadSelfGetThreadKernelPort$
$C2MachThreadSelfKernelPortNotDead$
$C3MachThreadSelfKernelPortNotNull$
$NotC4MachThreadSelfResourcesAvailable$
$MachThreadSelf\ Outputs$

$kernel\_port\_name! = Mach\_port\_null$

$RVMachThreadSelfKernelPortNull$

$C1MachThreadSelfGetThreadKernelPort$
$NotC3MachThreadSelfKernelPortNotNull$
$MachThreadSelf\ Outputs$

$kernel\_port\_name! = Mach\_port\_null$

$RVMachThreadSelfKernelPortDead$

$C1MachThreadSelfGetThreadKernelPort$
$NotC2MachThreadSelfKernelPortNotDead$
$MachThreadSelf\ Outputs$

$kernel\_port\_name! = Mach\_port\_dead$

$Return\_status\_to\_name : KERNEL\_RETURN \longrightarrow NAME$

$RVMachThreadSelfNoGetThreadKernelPort$

$NotC1MachThreadSelfGetThreadKernelPort$
$MachThreadSelf\ Outputs$

$kernel\_port\_name! = Return\_status\_to\_name(Kern\_insufficient\_permission)$

7.2.4 State Changes

A successful **mach_thread_self** request results in the addition of a send right for the kernel port, *port*, of *curr_th*?? into the port name space of the task, *task*, containing *curr_th*??. This is accomplished by schema *AddSendRight* which describes the relationships between *name*, *task*, and *port* when a send right is successfully added to a name space.

$$
\begin{array}{|l}
\hline
\textit{MachThreadSelfState} \\
\textit{Transition} \\
\textit{ThreadInvariants} \\
\Xi\ \textit{ThreadExist} \\
\Xi\ \textit{PortExist} \\
\Xi\ \textit{TaskExist} \\
\Xi\ \textit{DeadRights} \\
\Xi\ \textit{Threads} \\
\Xi\ \textit{SpecialThreadPorts} \\
\Xi\ \textit{ThreadAndProcessorSet} \\
\textit{AddSendRight} \\
\hline
task = curr\_task?? \\
port = thread\_sself(curr\_th??) \\
\\
\underline{p}ort\_set\_rel' = \underline{p}ort\_set\_rel \\
\hline
\end{array}
$$

7.2.5 Complete Request

Here we discuss the transitions shown in Figure 5 which describe the general form of the **mach_thread_self** request.

1. A **mach_thread_self** request is invoked through a system trap that has the $trap\_id?$ field set to $Mach\_thread\_self\_trap$.

$$
\begin{array}{|l}
\hline
\textit{InvokeMachThreadSelf} \\
\textit{Invoke} \\
\hline
trap\_id? = Mach\_thread\_self\_trap \\
\hline
\end{array}
$$

2. $MachThreadSelfPermCheckGTKP$ suspends processing to wait for the availability of a ruling on the $Get\_thread\_kernel\_port$ permission. The permission check is handled by the utilities described in Section 6.2 and consists of one or two transitions depending upon the availability of a ruling in the cache.

Editorial Note:
This should eventually be covered by generic trap processing if possible.

$$
\begin{array}{|l}
\hline
\textit{MachThreadSelfPermCheckGTKP} \\
\textit{Transition} \\
\hline
\exists\ CheckPending;\ user\_spec : UserSpecified \\
\bullet\ curr\_bk?? = Bk\_new\_trap(Mach\_thread\_self\_trap, user\_spec) \\
\quad \land\ ssi = thread\_sid(curr\_th??) \\
\quad \land\ osi = Thread\_self\_sid \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Get\_thread\_kernel\_port, E\_user(user\_spec)) \} \\
\hline
\end{array}
$$

3. Either $MachThreadSelfMiddleBad$ or $MachThreadSelfMiddleGood$ occurs. This models a transition where a ruling on the $Get\_thread\_kernel\_port$ permission has been obtained

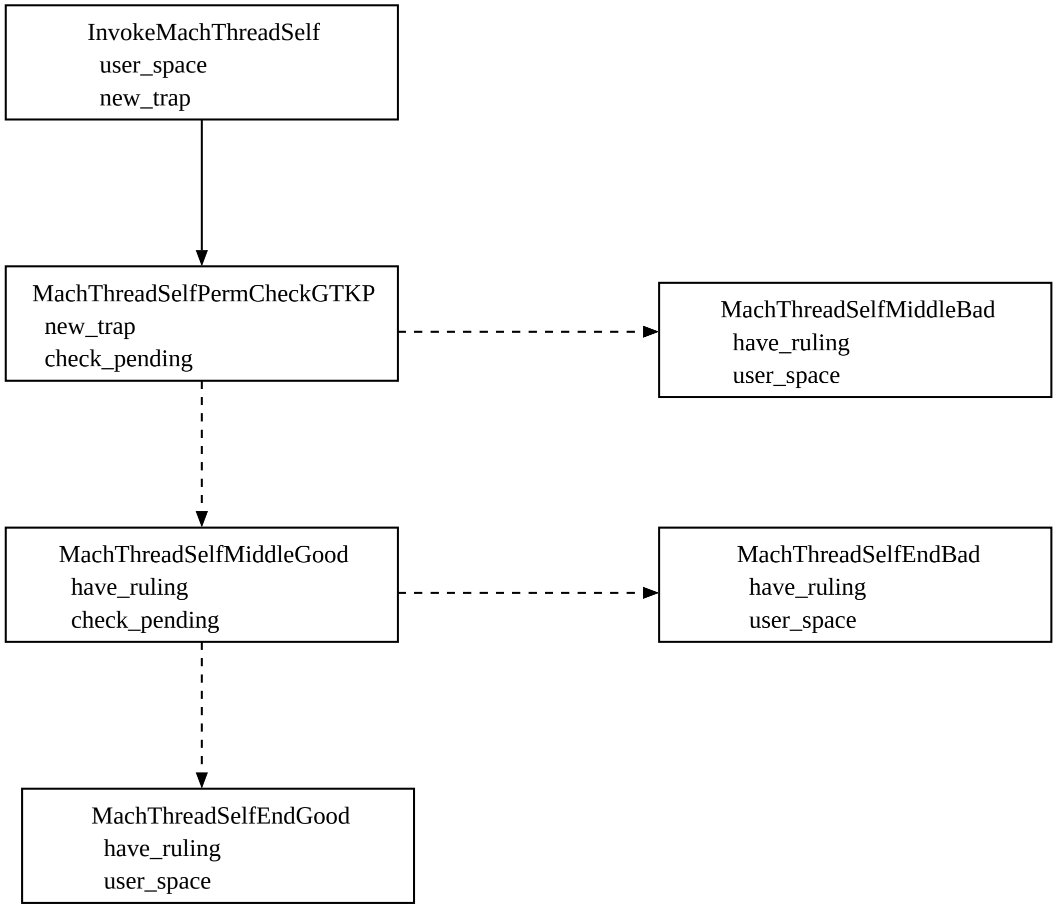

Figure 5: **mach_thread_self** Processing

from the cache or Security Server. Conditions **C1**–**C4** are checked. If any of them is false, no state changes are made, an error value is returned and processing terminates with this step.

$$
\begin{aligned}
& \mathit{MachThreadSelfMiddleBad} \\
& \quad \widehat{=} \; (\mathit{RVMachThreadSelfResourceShortage} \\
& \qquad \lor \mathit{RVMachThreadSelfKernelPortNull} \\
& \qquad \lor \mathit{RVMachThreadSelfKernelPortDead} \\
& \qquad \lor \mathit{RVMachThreadSelfNoGetThreadKernelPort}) \\
& \quad \land \mathit{TrapOnlyObserves}
\end{aligned}
$$

If the four conditions are all true, then trap processing is suspended (via $\mathit{MachThreadSelfPermCheckHS}$) while waiting to check on the availability of a ruling for the $\mathit{Hold\_send}$ permission.

Editorial Note:
The value used for osi in the following schema is not in agreement with the requirements in the FSPM which state that the osi should be $\mathit{port\_sid}(\mathit{port})$. However, this is the value that is being used in the prototype as of 17 May 1996. CAR# 5024 describes this discrepancy.

Also, env should not be a free variable but should be explicitly specified, in this case most likely as the "empty" environment.

$MachThreadSelfPermCheckHS$
$Transition$

$$\exists\, CheckPending;\ env : ENVIRONMENT;\ port : PORT$$
$$\bullet\ ssi = thread\_sid(curr\_th??)$$
$$\land\ port = thread\_sself(curr\_th??)$$
$$\land\ osi = \textbf{if}\ (curr\_th??, port) \in thread\_self\ \textbf{then}\ Thread\_self\_sid\ \textbf{else if}\ (curr\_task??, port) \in task\_self\ \textbf{then}\ Task\_self\_sid\ \textbf{else}\ port\_sid(port)$$
$$\land\ breaks' = breaks \oplus \{\ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Hold\_send, env)\ \}$$

$$MachThreadSelfMiddleGood \mathrel{\widehat=} C1MachThreadSelfGetThreadKernelPort \land C2MachThreadSelfKernelPortNotDead \land C3MachThreadSelfKernelPortNotNull \land C4MachThreadSelfResourcesAvailable \land MachThreadSelfPermCheckHS$$

4. Either $MachThreadSelfEndGood$ or $MachThreadSelfEndBad$ occurs. This models a transition where a ruling on the $Hold\_send$ permission has been obtained from the cache or the Security Server. The $Hold\_send$ permission is checked. If it is granted, the state changes in $MachThreadSelfState$ occur, and the name of the new send right is returned.

$$MachThreadSelfEndGood \mathrel{\widehat=} RVMachThreadSelfGood \land MachThreadSelfState$$

Otherwise, $Mach\_port\_null$ is returned and no state changes occur.

$$MachThreadSelfEndBad \mathrel{\widehat=} RVMachThreadSelfNoHoldSend \land TrapOnlyObserves$$

Section 8 Port Requests

8.1 Introduction to Port Requests

This chapter describes the port kernel requests in DTOS.

8.1.1 Request Identifiers

We first define the identifier that is used to represent each port request. The kernel accepts all port requests through task self ports.

$$
\begin{array}{l}
Mach\_port\_allocate\_id, Mach\_port\_allocate\_secure\_id, Mach\_port\_allocate\_name\_id, \\
\quad Mach\_port\_allocate\_name\_secure\_id, Mach\_port\_deallocate\_id, \\
\quad Mach\_port\_destroy\_id, Mach\_port\_extract\_right\_id, \\
\quad Mach\_port\_get\_receive\_status\_id, Mach\_port\_get\_refs\_id, \\
\quad Mach\_port\_get\_set\_status\_id, Mach\_port\_insert\_right\_id, \\
\quad Mach\_port\_mod\_refs\_id, Mach\_port\_move\_member\_id, Mach\_port\_names\_id, \\
\quad Mach\_port\_rename\_id, Mach\_port\_request\_notification\_id, \\
\quad Mach\_port\_set\_mscount\_id, Mach\_port\_set\_qlimit\_id, Mach\_port\_set\_seqno\_id, \\
\quad Mach\_port\_type\_id, Mach\_port\_type\_secure\_id : OPERATION \\
Port\_operations : \mathbb{P}\ OPERATION \\
\hline
\langle Mach\_port\_allocate\_id, Mach\_port\_allocate\_secure\_id, Mach\_port\_allocate\_name\_id, \\
\quad Mach\_port\_allocate\_name\_secure\_id, Mach\_port\_deallocate\_id, \\
\quad Mach\_port\_destroy\_id, Mach\_port\_extract\_right\_id, \\
\quad Mach\_port\_get\_receive\_status\_id, Mach\_port\_get\_refs\_id, \\
\quad Mach\_port\_get\_set\_status\_id, Mach\_port\_insert\_right\_id, \\
\quad Mach\_port\_mod\_refs\_id, Mach\_port\_move\_member\_id, Mach\_port\_names\_id, \\
\quad Mach\_port\_rename\_id, Mach\_port\_request\_notification\_id, \\
\quad Mach\_port\_set\_mscount\_id, Mach\_port\_set\_qlimit\_id, Mach\_port\_set\_seqno\_id, \\
\quad Mach\_port\_type\_id, Mach\_port\_type\_secure\_id\rangle \\
\qquad Values\_partition\ Port\_operations \\
Port\_operations \subseteq Allowed\_mach\_services(Pc\_task)
\end{array}
$$

8.1.2 Required Permissions

For each operation there is a primary permission that is required to perform the operation. We define here the portion of the $Required\_permission$ function that pertains to port requests.

$$\{(Mach\_port\_allocate\_id, Add\_name),$$
$$(Mach\_port\_get\_receive\_status\_id, Observe\_pns\_info),$$
$$(Mach\_port\_get\_refs\_id, Observe\_pns\_info),$$
$$(Mach\_port\_get\_set\_status\_id, Observe\_pns\_info),$$
$$(Mach\_port\_names\_id, Observe\_pns\_info),$$
$$(Mach\_port\_rename\_id, Port\_rename),$$
$$(Mach\_port\_set\_mscount\_id, Alter\_pns\_info),$$
$$(Mach\_port\_set\_qlimit\_id, Alter\_pns\_info),$$
$$(Mach\_port\_set\_seqno\_id, Alter\_pns\_info)\}$$
$$\subset Required\_permission$$

8.1.3 Invariant Information

Review Note:
This section will be completed in a future draft when all of the port requests are completed.

8.1.4 General Information

This section contains bits of information which are common to several port requests.

Review Note:
In a future draft, it may be appropriate to move some of this information to the state chapter.

The names $Mach\_port\_dead$ and $Mach\_port\_null$ are referred to in this chapter as $Reserved\_names$.

$$Reserved\_names : \mathbb{P}\, NAME$$
$$Reserved\_names = \{Mach\_port\_dead, Mach\_port\_null\}$$
$$Mach\_port\_null \neq Mach\_port\_dead$$

Parameters to several of the port requests include a type of right which includes not only send, send-once and receive rights but also dead rights and port sets. These parameters are given the type $RIGHT\_TYPE$.

$$[RIGHT\_TYPE]$$

$$Mach\_port\_right\_send, Mach\_port\_right\_receive, Mach\_port\_right\_send\_once,$$
$$Mach\_port\_right\_port\_set, Mach\_port\_right\_dead\_name : RIGHT\_TYPE$$
$$Values\_disjoint\langle Mach\_port\_right\_send, Mach\_port\_right\_receive,$$
$$Mach\_port\_right\_send\_once, Mach\_port\_right\_port\_set,$$
$$Mach\_port\_right\_dead\_name\rangle$$

The **mach_port_request_notification** request has a parameter of type $MACH\_MSG\_ID$ which indicates which of the three types of notification is being requested.

$$[MACH\_MSG\_ID]$$

$$\begin{array}{l} Mach\_notify\_port\_destroyed, Mach\_notify\_no\_senders, Mach\_notify\_dead\_name : MACH\_MSG\_ID \\ \hline Values\_disjoint\langle Mach\_notify\_port\_destroyed, Mach\_notify\_no\_senders, Mach\_notify\_dead\_name\rangle \end{array}$$

The **mach_port_get_receive_status** request returns a record $PortStatus$ consisting of the following information:

- $port\_set\_name$ — if the receive right is a member of a port set, the name of this port set; otherwise, the name $Mach\_port\_null$
- $make\_send\_count\_value$ — the make-send count for the port
- $port\_destroyed\_notification\_requested$ — takes on the value $True$ if a port-destroyed notification request is currently active for the port; otherwise $False$
- $no\_more\_senders\_notification\_requested$ — takes on the value $True$ if a no-more-senders notification request is currently active for the port; otherwise $False$
- msg_count_value — the number of messages queued at the port
- $qlimit\_value$ — the limit on the number of messages which can be queued to the port
- $sequence\_no\_value$ — the current sequence number for the port
- $number\_of\_send\_once\_rights$ — the number of send-once rights which exist to the port
- any_send_rights — takes on the value $True$ if there exist send rights to the port; otherwise $False$

$$\begin{array}{l} \textbf{PortStatus} \\ \hline port\_set\_name : NAME \\ make\_send\_count\_value : \mathbb{N} \\ port\_destroyed\_notification\_requested : BOOLEAN \\ no\_more\_senders\_notification\_requested : BOOLEAN \\ msg\_count\_value : \mathbb{N} \\ qlimit\_value : \mathbb{N} \\ sequence\_no\_value : \mathbb{Z} \\ number\_of\_send\_once\_rights : \mathbb{N} \\ any\_send\_rights : BOOLEAN \\ \hline \end{array}$$

The **mach_port_names**, **mach_port_type**, and **mach_port_type_secure** requests return a mask $PortTypeMask$ consisting of the following flags:

- $mach\_port\_type\_send$ — equal to $True$ if and only if the name is a send right
- $mach\_port\_type\_receive$ — equal to $True$ if and only if the name is a receive right
- $mach\_port\_type\_send\_once$ — equal to $True$ if and only if the name is a send-once right
- $mach\_port\_type\_port\_set$ — equal to $True$ if and only if the name is a port set
- $mach\_port\_type\_dead\_name$ — equal to $True$ if and only if the name is a dead name
- $mach\_port\_type\_dead\_name\_request$ — equal to $True$ if and only if there is an outstanding dead name notification request for the name
- $mach\_port\_type\_msg\_accepted\_request$ — equal to $True$ if and only if the name is not available for use as a send right since it has been used to force a message on a message queue

$$\begin{array}{l} \textit{PortTypeMask} \\ \hline mach\_port\_type\_send : BOOLEAN \\ mach\_port\_type\_receive : BOOLEAN \\ mach\_port\_type\_send\_once : BOOLEAN \\ mach\_port\_type\_port\_set : BOOLEAN \\ mach\_port\_type\_dead\_name : BOOLEAN \\ mach\_port\_type\_dead\_name\_request : BOOLEAN \\ mach\_port\_type\_msg\_accepted\_request : BOOLEAN \end{array}$$

8.1.5 Parameter Packaging Functions

When invoking a kernel request, the following functions package the input parameters into a message body:

$$\begin{array}{l} Mach\_port\_allocate\_inputs\_to\_body : RIGHT\_TYPE \longrightarrow MESSAGE\_BODY \\ Mach\_port\_extract\_right\_inputs\_to\_body \\ \quad : NAME \times MACH\_MSG\_TYPE \longrightarrow MESSAGE\_BODY \\ Mach\_port\_get\_receive\_status\_inputs\_to\_body : NAME \longrightarrow MESSAGE\_BODY \\ Mach\_port\_get\_refs\_inputs\_to\_body : NAME \times RIGHT\_TYPE \longrightarrow MESSAGE\_BODY \\ Mach\_port\_get\_set\_status\_inputs\_to\_body : NAME \longrightarrow MESSAGE\_BODY \\ Mach\_port\_insert\_right\_inputs\_to\_body \\ \quad : NAME \times NAME \times MACH\_MSG\_TYPE \longrightarrow MESSAGE\_BODY \\ Mach\_port\_move\_member\_inputs\_to\_body : NAME \times NAME \longrightarrow MESSAGE\_BODY \\ Mach\_port\_rename\_inputs\_to\_body : NAME \times NAME \longrightarrow MESSAGE\_BODY \\ Mach\_port\_request\_notification\_inputs\_to\_body \\ \quad : NAME \times MACH\_MSG\_ID \times \mathbb{N} \times NAME \times MACH\_MSG\_TYPE \longrightarrow MESSAGE\_BODY \\ Mach\_port\_set\_mscount\_inputs\_to\_body : NAME \times \mathbb{N} \longrightarrow MESSAGE\_BODY \\ Mach\_port\_set\_qlimit\_inputs\_to\_body : NAME \times \mathbb{N} \longrightarrow MESSAGE\_BODY \\ Mach\_port\_set\_seqno\_inputs\_to\_body : NAME \times \mathbb{N} \longrightarrow MESSAGE\_BODY \end{array}$$

When creating a reply message from a request, the following functions package the output parameters into a kernel reply:

$$\begin{array}{l} Mach\_port\_allocate\_outputs\_to\_reply : NAME \longrightarrow KERNEL\_REPLY \\ Mach\_port\_extract\_right\_outputs\_to\_reply \\ \quad : PORT \times MACH\_MSG\_TYPE \longrightarrow KERNEL\_REPLY \\ Mach\_port\_get\_receive\_status\_outputs\_to\_reply : PortStatus \longrightarrow KERNEL\_REPLY \\ Mach\_port\_get\_refs\_outputs\_to\_reply : \mathbb{N} \longrightarrow KERNEL\_REPLY \\ Mach\_port\_get\_set\_status\_outputs\_to\_reply : \mathbb{P}\, NAME \times \mathbb{N} \longrightarrow KERNEL\_REPLY \\ Mach\_port\_names\_outputs\_to\_reply \\ \quad : \mathrm{seq}\, NAME \times \mathbb{N} \times \mathrm{seq}\, PortTypeMask \times \mathbb{N} \longrightarrow KERNEL\_REPLY \\ Mach\_port\_request\_notification\_outputs\_to\_reply : PORT \longrightarrow KERNEL\_REPLY \end{array}$$

When receiving a reply message from the kernel the following functions unpack the message body to obtain the output parameters (including the return status):

$$
\begin{array}{l}
Body\_to\_mach\_port\_allocate\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times NAME \\
Body\_to\_mach\_port\_extract\_right\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times NAME \times MACH\_MSG\_TYPE \\
Body\_to\_mach\_port\_get\_receive\_status\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times PortStatus \\
Body\_to\_mach\_port\_get\_refs\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times \mathbb{N} \\
Body\_to\_mach\_port\_get\_set\_status\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times \mathbb{P}\, NAME \times \mathbb{N} \\
Body\_to\_mach\_port\_insert\_right\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \\
Body\_to\_mach\_port\_move\_member\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \\
Body\_to\_mach\_port\_names\_outputs \\
\quad : MESSAGE\_BODY \\
\qquad \longrightarrow KERNEL\_RETURN \times \mathrm{seq}\, NAME \times \mathbb{N} \times \mathrm{seq}\, PortTypeMask \times \mathbb{N} \\
Body\_to\_mach\_port\_rename\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \\
Body\_to\_mach\_port\_request\_notification\_outputs \\
\quad : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \times NAME \\
Body\_to\_mach\_port\_set\_mscount\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \\
Body\_to\_mach\_port\_set\_qlimit\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN \\
Body\_to\_mach\_port\_set\_seqno\_outputs : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN
\end{array}
$$

8.1.6 Kernel Processing

The initial kernel processing of any request, when removing the request from the bag of validated requests, is described by the $ProcessRequest$ schema in Section 6. In this section, we consider additional initial processing which is shared by all port requests.

The service port through which a port request is received must be the self port for some task. If it is not, then the request immediately terminates and returns $Kern\_invalid\_task$.

$$
\begin{array}{l}
\textbf{ProcessPortRequestBad} \\
\hline
ProcessRequest \\
SpecialTaskPorts \\
return! : KERNEL\_RETURN \\
\hline
operation? \in Port\_operations \\
service\_port? \notin \mathrm{dom}\, self\_task \\
return! = Kern\_invalid\_task
\end{array}
$$

$$PortRequestBad \mathrel{\widehat=} ProcessPortRequestBad \gg RequestNoOp$$

Otherwise, the task owning the self port is identified and the kernel continues to process the request.

$$\begin{array}{|l}
\multicolumn{1}{l}{__ ProcessPortRequestGood ____________________} \\
ProcessRequest \\
SpecialTaskPorts \\
task? : TASK \\
\hline
operation? \in Port_operations \\
service_port? \in \operatorname{dom} self_task \\
task? = self_task(service_port?) \\
\hline
\end{array}$$

Review Note:
Several of the port requests have the potential of returning $Kern\_resource\_shortage$. The model cannot accurately reflect when this is returned, so the original plans were to include another version of the "bad" schema here which dealt with those cases. However, this has not been done.

Review notes have been added to each of the individual requests for which $Kern\_resource\_shortage$ is a possible return value.

8.1.7 Notifications

Several of the port requests can result in the sending of notifications, messages which inform about some change to the state of a port or port right. There are six kinds of notifications:

dead-name A dead-name notification for a port right is registered by **mach_msg** or **mach_port_request_notification**. The notification is sent if the port right becomes dead due to the destruction of the port.

port-deleted A port-deleted notification is registered whenever a dead-name notification is registered. The notification is sent if the port right becomes unusable due to the right itself being destroyed or moved.

msg-accepted A msg-accepted notification for a port right is registered when the port right is used to forcibly enqueue a message on a port using **mach_msg**. The notification is sent when the message is removed from the queue (either to be received or destroyed).

no-senders A no-senders notification for a port is registered using **mach_port_request_notification**. The notification is sent when the last send or send-once right for the port is destroyed (with one exception depending upon the parameter $sync$ to **mach_port_request_notification**; see Section 8.8).

port-destroyed A port-destroyed notification for a port is registered using **mach_port_request_notification**. The notification is sent when the port would otherwise be destroyed, and contains a receive right for the port, thus saving it from destruction.

send-once A send-once notification for a port is sent whenever a send-once right is destroyed without being used to send a message.

Note that the kernel cannot guarantee to send notifications. It can fail to send a notification, for instance if it determines that an attempt to send the notification could result in an infinite loop.

Editorial Note:

Port-destroyed notifications could interact in interesting ways with the kernel's ability to identify "circularities" of receive rights in transit.

A cicrularity in receive rights exists when the receive right for port 1 is contained in a message destined for port 2, the receive right for port 2 is contained in a message destined for port 3, ... and the receive right for port N is contained in a message for port 1. If this occurs, then none of the messages can be received and the kernel sets out to clean them all up.

However, if the kernel cleans out a message and finds a receive right with a registered port-destroyed notification port, it instead sends the receive right to that port, saving the right from destruction. This breaks the circle, so other messages in the circle may once again be reachable.

We have not spent any time trying to determine how this interesting case is handled. It should be considered when port-destroyed notifications are modeled.

Sending of each kind of notification is modeled in the following sections.

Review Note:
For now, only dead-name notifications will be considered. When other types of notifications are considered, processing which is common to multiple notifications should be moved to this introductory section.

8.1.7.1 Dead Name Notifications The internal message header for a dead-name notification message is filled in as follows:

- The $local\_port$ and $local\_rights$ fields are empty since there is no expected reply from a dead-name notification.
- The $remote\_port$ and $remote\_rights$ fields contain a send-once right to the notification port.
- The $size$ field contains the size of a dead-name notification, which is a constant.
- The $msg\_sequence\_no$ field is initialized to zero. This field is ignored until the message is received.
- The $operation$ field contains the identifier $Ipc\_notify\_dead\_name\_id$.
- The $complex$ field is empty since the body of a dead-name notification contains no port rights or out-of-line memory.

$$
\begin{array}{|l}
Dead\_name\_notification\_header : PORT \longrightarrow MachInternalHeader \\
Dead\_name\_notification\_size : \mathbb{N} \\
\hline
\forall\, port : PORT \\
\bullet\ (\mathbf{let}\ header == Dead\_name\_notification\_header(port) \\
\qquad \bullet\ header.local\_port = \varnothing \\
\qquad \land\ header.local\_rights = \varnothing \\
\qquad \land\ header.remote\_port = port \\
\qquad \land\ header.remote\_rights = Mach\_msg\_type\_port\_send\_once \\
\qquad \land\ header.size = Dead\_name\_notification\_size \\
\qquad \land\ header.msg\_sequence\_no = 0 \\
\qquad \land\ header.operation = Ipc\_notify\_dead\_name\_id \\
\qquad \land\ header.complex = \varnothing)
\end{array}
$$

The body of a dead-name notification consists of a single element containing the name of the now dead port right.

Review Note:
I tried to specify the body of the message explicitly in terms of the model, but was unable to do so. The lack of a direct map between the model and the code caused some difficulty, but the final nail was the fact that the model requires a task to be associated with each message element, and this made no sense in the current situation.

$$Dead\_name\_notification\_body : NAME \longrightarrow INTERNAL\_BODY$$

The entire dead-name notification message contains a header and body as just described.

Review Note:
Once again, I would like to fill in the other values from the schema $InternalMessage$, but these values do not correspond to anything in the code and it is difficult to determine what values to choose.

$$Dead\_name\_notification\_message : PORT \times NAME \longrightarrow InternalMessage$$

$$\forall\, port : PORT;\ name : NAME$$
$$\bullet\ (Dead\_name\_notification\_message(port, name)).header = Dead\_name\_notification\_header(port)$$
$$\wedge\ (Dead\_name\_notification\_message(port, name)).body = Dead\_name\_notification\_body(name)$$

The following occurs when a dead name notification is sent:

- A new message ($new\_message$) is created, whose contents are give by $Dead\_name\_notification\_message(notify\_port, dead\_right\_name)$.
- $new\_message$ is added to the queue for $notify\_port$.
- The sending SID for $new\_message$ is set to reflect the kernel acting as $current\_task$.
- No receiving, sending SID or access vector is specified for the message.

Review Note:
Failure can occur here if the kernel is unable to allocate memory for the message. In this case the notification message is not sent and the kernel simply continues processing as if the message were successfully sent.

Review Note:
There is a permission check here which has not been specified. This check occurs after the message is allocated, however, in the current prototype the message is not deallocated when the permission check fails. The kernel simply continues processing as if the message were successfully sent.

This permission check will need to be added later when the execution model is updated. The $Ruling$ associated with the message will also need to be added at that time.

There is also a much bigger list of invariants that could be added to the following schema.

Review Note:
The code also has to make sure that $notify\_port$ is not Ip_dead. This should come for free whenever the schema is used in this chapter, but that should be doublechecked. If $notify\_port$ is Ip_dead, then the message is destroyed.

$$
\begin{array}{l}
\textit{QueueDeadNameNotification} \\
\Delta\ Messages \\
\Delta\ DtosMessages \\
KernelAs \\
current\_task : TASK \\
notify\_port : PORT \\
dead\_right\_name : NAME \\
\hline
\exists\, new\_message : MESSAGE \\
\bullet\ (new\_message \notin \underline{m}essage\_exists \\
\quad \land\ \underline{m}essage\_exists' = \underline{m}essage\_exists \cup \{new\_message\} \\
\quad \land\ \underline{m}sg\_contents' = \underline{m}sg\_contents \\
\qquad \oplus \{new\_message \mapsto Dead\_name\_notification\_message(notify\_port, dead\_right\_name)\} \\
\quad \land\ \underline{m}essage\_in\_port\_rel' = \underline{m}essage\_in\_port\_rel \\
\qquad \oplus \{notify\_port \mapsto \underline{m}essage\_in\_port\_rel(notify\_port) \frown \langle new\_message \rangle\} \\
\quad \land\ \underline{m}sg\_sending\_sid' = \underline{m}sg\_sending\_sid \\
\qquad \oplus \{new\_message \mapsto kernel\_as(current\_task)\}) \\
\underline{m}sg\_receiving\_sid' = \underline{m}sg\_receiving\_sid \\
\underline{m}sg\_specified\_sid' = \underline{m}sg\_specified\_sid \\
\underline{m}sg\_specified\_vector' = \underline{m}sg\_specified\_vector
\end{array}
$$

Review Note:
It's worth noting that the recipient of a dead-name notification only receives the name of the dead right, and no indication of which ipc name space contains the dead right.

8.2 mach_port_allocate

A **mach_port_allocate** request creates a new receive right, port set, or dead name in a task's name space.

8.2.1 Client Interface

kern_return_t **mach_port_allocate**
(mach_port_t *task_name*,
mach_port_right_t *right_type*,
mach_port_t* *new_name*);

kern_return_t **mach_port_allocate_secure**
(mach_port_t *task_name*,
mach_port_right_t *right_type*,
mach_port_t* *new_name*,
security_id_t *obj_sid*);

Review Note:
No attempt has been made to integrate **mach_port_allocate_secure** into this specification.

8.2.1.1 Input Parameters The following input parameters are provided by the client of a **mach_port_allocate** request:

- $task\_name?$ — the client's name for the task in whose name space the new receive right, port set or dead name is created
- $right\_type?$ — the type of right to be created, either $Mach\_port\_right\_receive$, $Mach\_port\_right\_port\_set$, or $Mach\_port\_right\_dead\_name$.

$$
\begin{array}{l}
\textit{MachPortAllocateClientInputs} \\
\hline
task\_name? : NAME \\
right\_type? : RIGHT\_TYPE \\
\hline
\end{array}
$$

A **mach_port_allocate** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_allocate\_id$ and has a body consisting of $right\_type?$.

$$
\begin{array}{l}
\textit{InvokeMachPortAllocate} \\
\hline
InvokeMachMsg \\
MachPortAllocateClientInputs \\
\hline
name? = task\_name? \\
operation? = Mach\_port\_allocate\_id \\
msg\_body = Mach\_port\_allocate\_inputs\_to\_body(right\_type?) \\
\hline
\end{array}
$$

8.2.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_allocate** request:

- $return!$ — the status of the request
- $new\_name!$ — the name of the new right

$$
\begin{array}{l}
\textit{MachPortAllocateClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
new\_name! : NAME \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{MachPortAllocateReceiveReply} \\
\hline
InvokeMachMsgRcv \\
MachPortAllocateClientOutputs \\
\hline
(return!, new\_name!) = Body\_to\_mach\_port\_allocate\_outputs(msg\_body) \\
\hline
\end{array}
$$

8.2.2 Kernel Interface

8.2.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_allocate** request:

- $task?$ — the task known to the client by $task\_name?$

- $right\_type?$ — provided by the client

$$
\begin{array}{|l}
\hline
\textit{MachPortAllocateInputs} \\
task? : TASK \\
right\_type? : RIGHT\_TYPE \\
\hline
\end{array}
$$

8.2.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_allocate** request:

- $return!$ — the status of the request
- $new\_name!$ — the name of the new right

$$
\begin{array}{|l}
\hline
\textit{MachPortAllocateOutputs} \\
return! : KERNEL\_RETURN \\
new\_name! : NAME \\
\hline
\end{array}
$$

Upon completion of the processing of a **mach_port_allocate** request, a reply message is built from the output parameters.

$$
\begin{array}{|l}
\hline
\textit{MachPortAllocateReply} \\
RequestReturn \\
new\_name? : NAME \\
\hline
reply? = Mach\_port\_allocate\_outputs\_to\_reply(new\_name?) \\
\hline
\end{array}
$$

8.2.3 Request Criteria

The following criteria are defined for the **mach_port_allocate** request:

- **C1** — $right\_type?$ is equal to one of the three values $Mach\_port\_right\_receive$, $Mach\_port\_right\_port\_set$, or $Mach\_port\_right\_dead\_name$.

$$
\begin{array}{|l}
\hline
\textit{C1MachPortAllocateRightIsValid} \\
right\_type? : RIGHT\_TYPE \\
\hline
right\_type? \in \{Mach\_port\_right\_receive, Mach\_port\_right\_port\_set, \\
\quad Mach\_port\_right\_dead\_name\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1MachPortAllocateRightIsValid \\
\quad \widehat{=} \neg\ C1MachPortAllocateRightIsValid
\end{array}
$$

- **C2** — $right\_type?$ is equal to $Mach\_port\_right\_receive$.

$$
\begin{array}{|l}
\hline
\textit{C2MachPortAllocateReceive} \\
right\_type? : RIGHT\_TYPE \\
\hline
right\_type? = Mach\_port\_right\_receive \\
\hline
\end{array}
$$

- **C3** — $right\_type?$ is equal to $Mach\_port\_right\_port\_set$.

$$\begin{array}{l} \textit{C3MachPortAllocatePortSet} \\ \hline right\_type? : RIGHT\_TYPE \\ \hline right\_type? = Mach\_port\_right\_port\_set \\ \hline \end{array}$$

- **C4** — $right\_type?$ is equal to $Mach\_port\_right\_dead\_name$.

$$\begin{array}{l} \textit{C4MachPortAllocateDeadName} \\ \hline right\_type? : RIGHT\_TYPE \\ \hline right\_type? = Mach\_port\_right\_dead\_name \\ \hline \end{array}$$

- **C5** — The number of rights belonging to $task?$ is less than the maximum.

Review Note:
The maximum number of rights per task used to be modeled by a constant value. This was not correct. However, the correct model has not been determined either. Therefore there is no predicate in the following schema. Note that this means that the schema $NotC5MachPortAllocateRoomInNameSpace$ is empty.

$$\begin{array}{l} \textit{C5MachPortAllocateRoomInNameSpace} \\ \hline PortNameSpace \\ task? : TASK \\ \hline \end{array}$$

$$NotC5MachPortAllocateRoomInNameSpace \\ \quad \widehat{=} \; PortNameSpace \land \neg \; C5MachPortAllocateRoomInNameSpace$$

8.2.4 Return Values

Table 2 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The table has been checked to agree with the code in CM as of 14Sep94.

Review Note:
This request can also return $Kern\_resource\_shortage$. If there were an attempt to model this, there would be three separate criteria for each of the three types of possible rights. Moreover, in the case when there is no room in the name space and no memory available for allocation, it appears that details in the current state of the name space may determine whether $Kern\_resource\_shortage$ or $Kern\_no\_space$ is returned.

Review Note:
According to the KID, there should also be a check that $task?$ has $Hold\_receive$ privilege to the new port whenever $right\_type?$=$mach\_port\_type\_receive$. This is not currently in the model nor was it in the code at the time the model was written.

$return!$	$new\_name!$	C1	C5
$Kern\_invalid\_value$	-	F	-
$Kern\_no\_space$	-	T	F
$Kern\_success$	$new\_name!$	T	T

Table 2: Return Values for **mach_port_allocate**

$$
\begin{array}{|l}
\hline
\textit{RVMachPortAllocateInvalidValue} \\
\textit{MachPortAllocateOutputs} \\
\textit{NotC1MachPortAllocateRightIsValid} \\
\hline
return! = Kern\_invalid\_value \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RVMachPortAllocateNoSpace} \\
\textit{MachPortAllocateOutputs} \\
\textit{C1MachPortAllocateRightIsValid} \\
\textit{NotC5MachPortAllocateRoomInNameSpace} \\
\hline
return! = Kern\_no\_space \\
\hline
\end{array}
$$

Review Note:
The way in which $new\_name!$ is actually determined is dependent upon the “next” available index in the name space and the generation number of that index. Since these details do not appear in the model, we cannot accurately model how $new\_name!$ is determined. Therefore all that is done here is to state properties on $new\_name!$.

$$
\begin{array}{|l}
\hline
\textit{RVMachPortAllocateSuccess} \\
\textit{MachPortAllocateOutputs} \\
\textit{C1MachPortAllocateRightIsValid} \\
\textit{C5MachPortAllocateRoomInNameSpace} \\
\hline
return! = Kern\_success \\
new\_name! \notin Reserved\_names \\
(task?, new\_name!) \notin local\_namep \\
\hline
\end{array}
$$

8.2.5 State Changes

Table 3 lists the possible successful executions of a **mach_port_allocate** request. Note that C2, C3 and C4 are mutually exclusive.

Case	C2	C3	C4	C5
$MachPortAllocateStateReceive$	T	-	-	T
$MachPortAllocateStatePortSet$	-	T	-	T
$MachPortAllocateStateDeadName$	-	-	T	T

Table 3: State Change Cases for **mach_port_allocate**

If the request is to allocate a new receive right, then

- a new port, $port!$, is created,
- $task?$ is given a receive right to $port!$ with the name $new\_name!$,
- the make-send count for $port!$ is initialized to 0,
- the queue size limit for $port!$ is initialized to the default value,
- the message queue for $port!$ is initialized to be empty,
- the sequence number for $port!$'s message queue is initialized to 0, and
- the SID for $port!$ is initialized to the default for $task?$.

$MachPortAllocateStateReceive$

$\Delta\ PortNameSpace$
$\Delta\ PortSummary$
$\Delta\ ObjectSid$
$PortSid$
$C2MachPortAllocateReceive$
$C5MachPortAllocateRoomInNameSpace$
$port! : PORT$
$new\_name! : NAME$

$\{port!\} = \underline{p}ort\_exists' \setminus \underline{p}ort\_exists$
$\underline{p}ort\_right\_rel' = \underline{p}ort\_right\_rel \cup \{(task?, port!, new\_name!, Receive, 1)\}$
$\underline{m}ake\_send\_count' = \underline{m}ake\_send\_count \oplus \{port! \mapsto 0\}$
$\underline{q}\_limit' = \underline{q}\_limit \oplus \{port! \mapsto Mach\_port\_q\_limit\_default\}$
$\underline{m}essage\_in\_port\_rel' = \underline{m}essage\_in\_port\_rel \oplus \{port! \mapsto \langle\rangle\}$
$\underline{s}equence\_no' = \underline{s}equence\_no \oplus \{port! \mapsto 0\}$
$\underline{p}ort\_sid' = \underline{p}ort\_sid \oplus \{port! \mapsto Default\_port\_sid(\underline{t}ask\_sid(task?))\}$

If the request is to allocate a new port set, then an empty port set with the name $new\_name!$ is added to $task?$'s name space.

$MachPortAllocateStatePortSet$

$\Delta\ PortNameSpace$
$C3MachPortAllocatePortSet$
$C5MachPortAllocateRoomInNameSpace$
$new\_name! : NAME$

$\underline{p}ort\_set\_rel' = \underline{p}ort\_set\_rel \cup \{(task?, new\_name!, \varnothing)\}$

If the request is to allocate a new dead right, then a dead right with the name $new\_name!$ and a user reference count of 1 is added to $task?$'s name space.

$MachPortAllocateStateDeadName$

$\Delta\ PortNameSpace$
$C4MachPortAllocateDeadName$
$C5MachPortAllocateRoomInNameSpace$
$new\_name! : NAME$

$\underline{d}ead\_right\_rel' = \underline{d}ead\_right\_rel \cup \{(task?, new\_name!, 1)\}$

Review Note:
Invariants should be stated here as well.

8.2.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_allocate** request is described in Section 8.1.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProcessingMachPortAllocate}} \\
\hline
\textit{ProcessPortRequestGood} \\
\hline
\textit{operation?} = \textit{Mach_port_allocate_id} \\
\hline
\end{array}$$

An unsuccessful **mach_port_allocate** request results in no changes to the Mach state and returns only the appropriate error status.

$$\begin{aligned}
&\textit{MachPortAllocateBad} \\
&\quad \widehat{=} (\textit{RVMachPortAllocateInvalidValue} \\
&\qquad \lor \textit{RVMachPortAllocateNoSpace}) \\
&\quad \gg \textit{RequestNoOp}
\end{aligned}$$

A successful **mach_port_allocate** request alters the Mach state as described in Section 8.2.5 and returns a reply message.

$$\begin{aligned}
&\textit{MachPortAllocateGood} \\
&\quad \widehat{=} ((\textit{MachPortAllocateStateReceive} \\
&\qquad \lor \textit{MachPortAllocateStatePortSet} \\
&\qquad \lor \textit{MachPortAllocateStateDeadName}) \\
&\quad \land \textit{RVMachPortAllocateSuccess}) \\
&\quad \gg \textit{MachPortAllocateReply}
\end{aligned}$$

The complete specification of kernel processing of a **mach_port_allocate** request consists of the initial processing followed by an unsuccessful or successful execution.

$$\begin{aligned}
&\textit{MachPortAllocate} \\
&\quad \widehat{=} \textit{ProcessingMachPortAllocate} \\
&\quad \mathbin{;} (\textit{MachPortAllocateBad} \\
&\qquad \lor \textit{MachPortAllocateGood})
\end{aligned}$$

8.3 mach_port_get_receive_status

A **mach_port_get_receive_status** request returns the current status of the port associated with a receive right.

8.3.1 Client Interface

```
kern_return_t mach_port_get_receive_status
              (mach_port_t           task_name,
               mach_port_t           right_name,
               mach_port_status_t*   port_status);
```

8.3.1.1 Input Parameters The following input parameters are provided by the client of a **mach_port_get_receive_status** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of a receive right for the port whose status is requested

$$
\begin{array}{l}
\textit{MachPortGetReceiveStatusClientInputs} \\
\hline
task\_name? : NAME \\
right\_name? : NAME \\
\hline
\end{array}
$$

A **mach_port_get_receive_status** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_get\_receive\_status\_id$ and has a body consisting of $right\_name?$.

$$
\begin{array}{l}
\textit{InvokeMachPortGetReceiveStatus} \\
\hline
InvokeMachMsg \\
MachPortGetReceiveStatusClientInputs \\
\hline
name? = task\_name? \\
operation? = Mach\_port\_get\_receive\_status\_id \\
msg\_body = Mach\_port\_get\_receive\_status\_inputs\_to\_body(right\_name?) \\
\hline
\end{array}
$$

8.3.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_get_receive_status** request:

- $return!$ — the status of the request
- $port\_status!$ — the status information for $right\_name?$, as described in Section 8.1.4.

$$
\begin{array}{l}
\textit{MachPortGetReceiveStatusClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
port\_status! : PortStatus \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{MachPortGetReceiveStatusReceiveReply} \\
\hline
InvokeMachMsgRcv \\
MachPortGetReceiveStatusClientOutputs \\
\hline
(return!, port\_status!) = Body\_to\_mach\_port\_get\_receive\_status\_outputs(msg\_body) \\
\hline
\end{array}
$$

8.3.2 Kernel Interface

8.3.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_get_receive_status** request:

- $task?$ — the task known to the client by $task\_name?$
- $right\_name?$ — provided by the client

$$
\begin{array}{l}
\textit{MachPortGetReceiveStatusInputs} \\
\hline
task? : TASK \\
right\_name? : NAME \\
\hline
\end{array}
$$

8.3.2.2 Output Parameters

The following output parameters are returned by the kernel for a **mach_port_get_receive_status** request:

- $return!$ — the status of the request
- $port\_status!$ — the status information for $right\_name?$, as described above

$$\begin{array}{l} \textit{MachPortGetReceiveStatusOutputs} \\ \hline return! : KERNEL\_RETURN \\ port\_status! : PortStatus \\ \hline \end{array}$$

Upon completion of the processing of a **mach_port_get_receive_status** request, a reply message is built from the output parameters.

$$\begin{array}{l} \textit{MachPortGetReceiveStatusReply} \\ \hline RequestOnlyObserves \\ port\_status? : PortStatus \\ \hline reply? = Mach\_port\_get\_receive\_status\_outputs\_to\_reply(port\_status?) \\ \hline \end{array}$$

8.3.3 Request Criteria

The following criteria are defined for the **mach_port_get_receive_status** request:

- **C1** — $right\_name?$ represents a right in $task?$'s name space.

$$\begin{array}{l} \textit{C1MachPortGetReceiveStatusNameIsARight} \\ \hline PortNameSpace \\ task? : TASK \\ right\_name? : NAME \\ \hline (task?, right\_name?) \in local\_namep \\ \hline \end{array}$$

$$NotC1MachPortGetReceiveStatusNameIsARight \\ \quad \widehat{=} PortNameSpace \land \neg\, C1MachPortGetReceiveStatusNameIsARight$$

- **C2** — $right\_name?$ represents a receive right in $task?$'s name space.

$$\begin{array}{l} \textit{C2MachPortGetReceiveStatusNameIsAReceiveRight} \\ \hline PortNameSpace \\ task? : TASK \\ right\_name? : NAME \\ \hline (task?, right\_name?) \in r\_right \\ \hline \end{array}$$

$$NotC2MachPortGetReceiveStatusNameIsAReceiveRight \\ \quad \widehat{=} PortNameSpace \land \neg\, C2MachPortGetReceiveStatusNameIsAReceiveRight$$

8.3.4 Return Values

Table 4 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The order of these checks agrees with the code in CM on 14Sep94.

$return!$	$port\_status!$	C1	C2
$Kern\_invalid\_name$	-	F	-
$Kern\_invalid\_right$	-	T	F
$Kern\_success$	as described below	T	T

Table 4: Return Values for **mach_port_get_receive_status**

$RVMachPortGetReceiveStatusInvalidName$
$MachPortGetReceiveStatus\ Outputs$
$NotC1MachPortGetReceiveStatusNameIsARight$

$return! = Kern\_invalid\_name$

$RVMachPortGetReceiveStatusInvalidRight$
$MachPortGetReceiveStatus\ Outputs$
$C1MachPortGetReceiveStatusNameIsARight$
$NotC2MachPortGetReceiveStatusNameIsAReceiveRight$

$return! = Kern\_invalid\_right$

In the successful case when $right\_name?$ refers to a receive right in $task?$'s name space, then the request returns a record with the following fields. Here $port?$ refers to the port named by $right\_name?$.

- $port\_set\_name$ — if $right\_name?$ is a member of a port set, the name of this port set; otherwise, the name $Mach\_port\_null$
- $make\_send\_count\_value$ — the make-send count for $port?$
- $port\_destroyed\_notification\_requested$ — a boolean value indicating if a port-destroyed notification request is currently active for $port?$
- $no\_more\_senders\_notification\_requested$ — a boolean value indicating if a no-more-senders notification request is currently active for $port?$
- msg_count_value — the number of messages queued at $port?$
- $qlimit\_value$ — the limit on the number of messages which can be queued at $port?$
- $sequence\_no\_value$ — the current sequence number for the message queue at $port?$
- $number\_of\_send\_once\_rights$ — the number of send-once rights which exist to $port?$
- any_send_rights — a boolean value indicating if there are existing send rights to $port?$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{RVMachPortGetReceiveStatusSuccess} \_\_\_\_} \\
\mathit{MachPortGetReceiveStatusOutputs} \\
\mathit{C1MachPortGetReceiveStatusNameIsARight} \\
\mathit{C2MachPortGetReceiveStatusNameIsAReceiveRight} \\
\mathit{PortSummary} \\
\mathit{Notifications} \\
\hline
\mathit{return!} = \mathit{Kern\_success} \\
(\mathbf{let}\ \mathit{port?} == \mathit{named\_port}(\mathit{task?}, \mathit{right\_name?}) \\
\quad \bullet\ \mathit{port\_status!.port\_set\_name} = \mathbf{if}\ \mathit{port?} \in \bigcup(\mathrm{ran}\ \mathit{port\_set}) \\
\qquad \mathbf{then}\ \mathit{containing\_set}(\mathit{port?}) \\
\qquad \mathbf{else}\ \mathit{Mach\_port\_null} \\
\quad \land\ \mathit{port\_status!.make\_send\_count\_value} = \underline{m}\mathit{ake\_send\_count}(\mathit{port?}) \\
\quad \land\ (\mathit{port\_status!.port\_destroyed\_notification\_requested} = \mathit{True} \\
\qquad \Leftrightarrow \mathit{port?} \in \mathrm{dom}\ \mathit{port\_notify\_destroyed}) \\
\quad \land\ (\mathit{port\_status!.no\_more\_senders\_notification\_requested} = \mathit{True} \\
\qquad \Leftrightarrow \mathit{port?} \in \mathrm{dom}\ \mathit{port\_notify\_no\_more\_senders}) \\
\quad \land\ \mathit{port\_status!.msg\_count\_value} = \mathit{port\_size}(\mathit{port?}) \\
\quad \land\ \mathit{port\_status!.qlimit\_value} = \underline{q}\mathit{\_limit}(\mathit{port?}) \\
\quad \land\ \mathit{port\_status!.sequence\_no\_value} = \underline{s}\mathit{equence\_no}(\mathit{port?}) \\
\quad \land\ (\mathit{port\_status!.number\_of\_send\_once\_rights} \\
\qquad = \#\{\mathit{task} : \mathit{TASK};\ \mathit{name} : \mathit{NAME};\ i : \mathbb{N}_1 \\
\qquad\quad \mid (\mathit{task}, \mathit{port?}, \mathit{name}, \mathit{Send\_once}, i) \in \underline{p}\mathit{ort\_right\_rel} \\
\qquad\quad \bullet\ \mathit{name}\}) \\
\quad \land\ (\mathit{port\_status!.any\_send\_rights} = \mathit{True} \\
\qquad \Leftrightarrow (\exists\ \mathit{task} : \mathit{TASK};\ \mathit{name} : \mathit{NAME};\ i : \mathbb{N}_1 \\
\qquad\quad \bullet\ (\mathit{task}, \mathit{port?}, \mathit{name}, \mathit{Send}, i) \in \underline{p}\mathit{ort\_right\_rel}))) \\
\hline
\end{array}
$$

Review Note:
The values of $\mathit{port\_status!.number\_of\_send\_once\_rights}$ and $\mathit{port\_status!.any\_send\_rights}$ may be incorrect in the specification, since they only count rights which currently exist in some name space. It is unclear whether the code also counts rights in transit.

8.3.5 State Changes

A **mach_port_get_receive_status** request does not make any state changes since it only observes the system state.

8.3.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_get_receive_status** request is described in Section 8.1.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{ProcessingMachPortGetReceiveStatus} \_\_\_\_} \\
\mathit{ProcessPortRequestGood} \\
\hline
\mathit{operation?} = \mathit{Mach\_port\_get\_receive\_status\_id} \\
\hline
\end{array}
$$

An unsuccessful **mach_port_get_receive_status** request results in no changes to the Mach state and returns only the appropriate error status.

$$\begin{aligned}&MachPortGetReceiveStatusBad\\&\quad \mathrel{\widehat=} (RVMachPortGetReceiveStatusInvalidName\\&\qquad \lor RVMachPortGetReceiveStatusInvalidRight)\\&\quad \gg RequestNoOp\end{aligned}$$

A successful **mach_port_get_receive_status** request results in no changes to the Mach state and returns a reply message.

$$\begin{aligned}&MachPortGetReceiveStatusGood\\&\quad \mathrel{\widehat=} RVMachPortGetReceiveStatusSuccess\\&\quad \gg MachPortGetReceiveStatusReply\end{aligned}$$

The complete specification of kernel processing of a **mach_port_get_receive_status** request consists of the initial processing followed by an unsuccessful or successful execution.

$$\begin{aligned}&MachPortGetReceiveStatus\\&\quad \mathrel{\widehat=} ProcessingMachPortGetReceiveStatus\\&\quad \fatsemi (MachPortGetReceiveStatusBad\\&\qquad \lor MachPortGetReceiveStatusGood)\end{aligned}$$

8.4 mach_port_get_refs

A **mach_port_get_refs** request returns the number of user references a task has for a right.

8.4.1 Client Interface

```
kern_return_t mach_port_get_refs
              (mach_port_t               task_name,
               mach_port_t               right_name,
               mach_port_right_t         right_type,
               mach_port_urefs_t*        refs);
```

8.4.1.1 Input Parameters

The following input parameters are provided by the client of a **mach_port_get_refs** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of the right whose reference count is desired
- $right\_type?$ — the type of right for which the reference count is requested, either $Mach\_port\_right\_send$, $Mach\_port\_right\_receive$, $Mach\_port\_right\_send\_once$, $Mach\_port\_right\_port\_set$, or $Mach\_port\_right\_dead\_name$

$$\begin{aligned}&\textit{MachPortGetRefsClientInputs}\\&task\_name? : NAME\\&right\_name? : NAME\\&right\_type? : RIGHT\_TYPE\end{aligned}$$

A **mach_port_get_refs** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_get\_refs\_id$ and has a body consisting of $right\_name?$ and $right\_type?$.

$$
\begin{array}{l}
\textit{InvokeMachPortGetRefs} \\
\hline
\textit{InvokeMachMsg} \\
\textit{MachPortGetRefsClientInputs} \\
\hline
name? = task\_name? \\
operation? = Mach\_port\_get\_refs\_id \\
msg\_body = Mach\_port\_get\_refs\_inputs\_to\_body(right\_name?, right\_type?)
\end{array}
$$

8.4.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_get_refs** request:

- $return!$ — the status of the request
- $refs!$ — the number of user references to the type of right indicated by $right\_type?$ and associated with $right\_name?$

$$
\begin{array}{l}
\textit{MachPortGetRefsClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
refs! : \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textit{MachPortGetRefsReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{MachPortGetRefsClientOutputs} \\
\hline
(return!, refs!) = Body\_to\_mach\_port\_get\_refs\_outputs(msg\_body)
\end{array}
$$

8.4.2 Kernel Interface

8.4.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_get_refs** request:

- $task?$ — the task known to the client by $task\_name?$
- $right\_name?$ — provided by the client
- $right\_type?$ — provided by the client

$$
\begin{array}{l}
\textit{MachPortGetRefsInputs} \\
\hline
task? : TASK \\
right\_name? : NAME \\
right\_type? : RIGHT\_TYPE
\end{array}
$$

8.4.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_get_refs** request:

- $return!$ — the status of the request
- $refs!$ — the number of user references for $right\_type?$ associated with $right\_name?$

$$\textit{MachPortGetRefsOutputs}$$
$$return! : KERNEL\_RETURN$$
$$refs! : \mathbb{N}$$

Upon completion of the processing of a **mach_port_get_refs** request, a reply message is built from the output parameters.

$$\textit{MachPortGetRefsReply}$$
$$RequestOnlyObserves$$
$$refs? : \mathbb{N}$$
$$reply? = Mach\_port\_get\_refs\_outputs\_to\_reply(refs?)$$

8.4.3 Request Criteria

The following criteria are defined for the **mach_port_get_refs** request:

- **C1** — The value of $right\_type?$ is one of the five values $Mach\_port\_right\_send$, $Mach\_port\_right\_receive$, $Mach\_port\_right\_send\_once$, $Mach\_port\_right\_port\_set$, or $Mach\_port\_right\_dead\_name$.

$$\textit{C1MachPortGetRefsRightIsRecognized}$$
$$right\_type? : RIGHT\_TYPE$$
$$right\_type? \in \{Mach\_port\_right\_send, Mach\_port\_right\_receive, Mach\_port\_right\_send\_once, Mach\_port\_right\_port\_set, Mach\_port\_right\_dead\_name\}$$

$$NotC1MachPortGetRefsRightIsRecognized \mathrel{\widehat=} \neg\ C1MachPortGetRefsRightIsRecognized$$

- **C2** — $right\_name?$ represents a right in $task?$'s name space.

$$\textit{C2MachPortGetRefsNameIsARight}$$
$$PortNameSpace$$
$$task? : TASK$$
$$right\_name? : NAME$$
$$(task?, right\_name?) \in local\_namep$$

$$NotC2MachPortGetRefsNameIsARight \mathrel{\widehat=} PortNameSpace \land \neg\ C2MachPortGetRefsNameIsARight$$

- **C3** — $right\_type?$ and $right\_name?$ both refer to a send right.

$$
\begin{array}{l}
\textbf{schema } C3MachPortGetRefsSend \\
\quad PortNameSpace \\
\quad task? : TASK \\
\quad right\_name? : NAME \\
\quad right\_type? : RIGHT\_TYPE \\
\hline
\quad right\_type? = Mach\_port\_right\_send \\
\quad (task?, right\_name?) \in s\_right
\end{array}
$$

$$
\begin{array}{l}
NotC3MachPortGetRefsSend \\
\quad \widehat{=}\ PortNameSpace \land \neg\ C3MachPortGetRefsSend
\end{array}
$$

- **C4** — $right\_type?$ and $right\_name?$ both refer to a dead name.

$$
\begin{array}{l}
\textbf{schema } C4MachPortGetRefsDeadName \\
\quad PortNameSpace \\
\quad task? : TASK \\
\quad right\_name? : NAME \\
\quad right\_type? : RIGHT\_TYPE \\
\hline
\quad right\_type? = Mach\_port\_right\_dead\_name \\
\quad (task?, right\_name?) \in dead\_namep
\end{array}
$$

$$
\begin{array}{l}
NotC4MachPortGetRefsDeadName \\
\quad \widehat{=}\ PortNameSpace \land \neg\ C4MachPortGetRefsDeadName
\end{array}
$$

- **C5** — $right\_type?$ and $right\_name?$ both refer to a receive right, send-once right, or port set.

$$
\begin{array}{l}
\textbf{schema } C5MachPortGetRefsOther \\
\quad PortNameSpace \\
\quad task? : TASK \\
\quad right\_name? : NAME \\
\quad right\_type? : RIGHT\_TYPE \\
\hline
\quad (right\_type? = Mach\_port\_right\_receive \\
\qquad \land (task?, right\_name?) \in r\_right) \\
\quad \lor (right\_type? = Mach\_port\_right\_send\_once \\
\qquad \land (task?, right\_name?) \in so\_right) \\
\quad \lor (right\_type? = Mach\_port\_right\_port\_set \\
\qquad \land (task?, right\_name?) \in port\_set\_namep)
\end{array}
$$

$$
\begin{array}{l}
NotC5MachPortGetRefsOther \\
\quad \widehat{=}\ PortNameSpace \land \neg\ C5MachPortGetRefsOther
\end{array}
$$

8.4.4 Return Values

Table 5 describes the values returned at the completion of the request and the conditions under which each value is returned. Note that criteria C3 through C5 are mutually exclusive by definition.

Review Note:
The order of the checks agree with the code in CM as of 14Sep94.

$return!$	$refs!$	C1	C2	C3	C4	C5
$Kern\_invalid\_value$	-	F	-	-	-	-
$Kern\_invalid\_name$	-	T	F	-	-	-
$Kern\_success$	0	T	T	F	F	F
$Kern\_success$	$s\_right\_ref\_count(task?, right\_name?)$	T	T	T	-	-
$Kern\_success$	$dead\_right\_ref\_count(task?, right\_name?)$	T	T	-	T	-
$Kern\_success$	1	T	T	-	-	T

Table 5: Return Values for **mach_port_get_refs**

$RVMachPortGetRefsInvalidValue$
$MachPortGetRefsOutputs$
$NotC1MachPortGetRefsRightIsRecognized$

$return! = Kern\_invalid\_value$

$RVMachPortGetRefsInvalidName$
$MachPortGetRefsOutputs$
$C1MachPortGetRefsRightIsRecognized$
$NotC2MachPortGetRefsNameIsARight$

$return! = Kern\_invalid\_name$

If $right\_name?$ does not represent a right of type $right\_type?$, the value 0 is returned.

$RVMachPortGetRefsWrongRight$
$MachPortGetRefsOutputs$
$C1MachPortGetRefsRightIsRecognized$
$C2MachPortGetRefsNameIsARight$
$NotC3MachPortGetRefsSend$
$NotC4MachPortGetRefsDeadName$
$NotC5MachPortGetRefsOther$

$return! = Kern\_success$
$refs! = 0$

If $right\_type? = Mach\_port\_right\_send$ and $right\_name?$ is a send right, then the number of user references to the send right is returned.

$$\begin{array}{l}
\textit{RVMachPortGetRefsSend} \\
\hline
\textit{MachPortGetRefsOutputs} \\
\textit{C1MachPortGetRefsRightIsRecognized} \\
\textit{C2MachPortGetRefsNameIsARight} \\
\textit{C3MachPortGetRefsSend} \\
\hline
return! = Kern_success \\
refs! = s_right_ref_count(task?, right_name?)
\end{array}$$

If $right\_type? = Mach\_port\_right\_dead\_name$ and $right\_name?$ is a dead right, then the number of user references to the dead right is returned.

$$\begin{array}{l}
\textit{RVMachPortGetRefsDeadName} \\
\hline
\textit{MachPortGetRefsOutputs} \\
\textit{C1MachPortGetRefsRightIsRecognized} \\
\textit{C2MachPortGetRefsNameIsARight} \\
\textit{C4MachPortGetRefsDeadName} \\
\hline
return! = Kern_success \\
refs! = dead_right_ref_count(task?, right_name?)
\end{array}$$

If $right\_type?$ and $right\_name?$ both refer to a receive right, send-once right, or port set, then the value 1 is returned since there is only one receive right, send-once right or port set associated with any name.

$$\begin{array}{l}
\textit{RVMachPortGetRefsOther} \\
\hline
\textit{MachPortGetRefsOutputs} \\
\textit{C1MachPortGetRefsRightIsRecognized} \\
\textit{C2MachPortGetRefsNameIsARight} \\
\textit{C5MachPortGetRefsOther} \\
\hline
return! = Kern_success \\
refs! = 1
\end{array}$$

8.4.5 State Changes

A **mach_port_get_refs** request does not make any state changes since it only observes the system state.

8.4.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_get_refs** request is described in Section 8.1.

$$\begin{array}{l}
\textit{ProcessingMachPortGetRefs} \\
\hline
\textit{ProcessPortRequestGood} \\
\hline
operation? = Mach_port_get_refs_id
\end{array}$$

An unsuccessful **mach_port_get_refs** request results in no changes to the Mach state and returns only the appropriate error status.

$$
\begin{aligned}
&MachPortGetRefsBad \\
&\quad \widehat{=} (RVMachPortGetRefsInvalidValue \\
&\qquad \vee RVMachPortGetRefsInvalidValue) \\
&\quad \gg RequestNoOp
\end{aligned}
$$

A successful **mach_port_get_refs** request results in no changes to the Mach state and returns a reply message.

$$
\begin{aligned}
&MachPortGetRefsGood \\
&\quad \widehat{=} (RVMachPortGetRefsWrongRight \\
&\qquad \vee RVMachPortGetRefsSend \\
&\qquad \vee RVMachPortGetRefsDeadName \\
&\qquad \vee RVMachPortGetRefsOther) \\
&\quad \gg MachPortGetRefsReply
\end{aligned}
$$

The complete specification of kernel processing of a **mach_port_get_refs** request consists of the initial processing followed by an unsuccessful or successful execution.

$$
\begin{aligned}
&MachPortGetRefs \\
&\quad \widehat{=} ProcessingMachPortGetRefs \\
&\quad ⨾ (MachPortGetRefsBad \\
&\qquad \vee MachPortGetRefsGood)
\end{aligned}
$$

8.5 mach_port_get_set_status

A **mach_port_get_set_status** request returns the names of the members of a given port set.

8.5.1 Client Interface

```
kern_return_t mach_port_get_set_status
                (mach_port_t                    task_name,
                 mach_port_t                    right_name,
                 mach_port_array_t*             member_names,
                 mach_msg_type_number_t*        count);
```

8.5.1.1 Input Parameters

The following input parameters are provided by the client of a **mach_port_get_set_status** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of the port set whose members are returned

MachPortGetSetStatusClientInputs

$$
\begin{aligned}
&task\_name? : NAME \\
&right\_name? : NAME
\end{aligned}
$$

A **mach_port_get_set_status** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_get\_set\_status\_id$ and has a body consisting of $right\_name?$.

$$
\begin{array}{l}
\textit{InvokeMachPortGetSetStatus} \\
\hline
\textit{InvokeMachMsg} \\
\textit{MachPortGetSetStatusClientInputs} \\
\hline
name? = task\_name? \\
operation? = Mach\_port\_get\_set\_status\_id \\
msg\_body = Mach\_port\_get\_set\_status\_inputs\_to\_body(right\_name?)
\end{array}
$$

8.5.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_get_set_status** request:

- $return!$ — the status of the request
- $member\_names!$ — the names of the members of the port set $right\_name?$
- $count!$ — the number of members of the port set $right\_name?$

$$
\begin{array}{l}
\textit{MachPortGetSetStatusClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
member\_names! : \mathbb{P}\ NAME \\
count! : \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textit{MachPortGetSetStatusReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{MachPortGetSetStatusClientOutputs} \\
\hline
(return!, member\_names!, count!) \\
\quad = Body\_to\_mach\_port\_get\_set\_status\_outputs(msg\_body)
\end{array}
$$

8.5.2 Kernel Interface

8.5.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_get_set_status** request:

- $task?$ — the task known to the client by $task\_name?$
- $right\_name?$ — provided by the client

$$
\begin{array}{l}
\textit{MachPortGetSetStatusInputs} \\
\hline
task? : TASK \\
right\_name? : NAME
\end{array}
$$

8.5.2.2 Output Parameters

The following output parameters are returned by the kernel for a **mach_port_get_set_status** request:

- $return!$ — the status of the request
- $member\_names!$ — the names of the members of the port set $right\_name?$
- $count!$ — the number of members of the port set $right\_name?$

$MachPortGetSetStatusOutputs$

$$return! : KERNEL\_RETURN$$
$$member\_names! : \mathbb{P}\ NAME$$
$$count! : \mathbb{N}$$

Upon completion of the processing of a **mach_port_get_set_status** request, a reply message is built from the output parameters.

$MachPortGetSetStatusReply$

$$RequestOnlyObserves$$
$$member\_names? : \mathbb{P}\ NAME$$
$$count? : \mathbb{N}$$
$$\overline{}$$
$$reply? = Mach\_port\_get\_set\_status\_outputs\_to\_reply(member\_names?, count?)$$

8.5.3 Request Criteria

The following criteria are defined for the **mach_port_get_set_status** request:

- **C1** — $right\_name?$ represents a right in $task?$'s name space.

$C1MachPortGetSetStatusNameIsARight$

$$PortNameSpace$$
$$task? : TASK$$
$$right\_name? : NAME$$
$$\overline{}$$
$$(task?, right\_name?) \in local\_namep$$

$$NotC1MachPortGetSetStatusNameIsARight$$
$$\mathrel{\widehat=} PortNameSpace \land \lnot\ C1MachPortGetSetStatusNameIsARight$$

- **C2** — $right\_name?$ represents a port set in $task?$'s name space.

$C2MachPortGetSetStatusNameIsAPortSet$

$$PortNameSpace$$
$$task? : TASK$$
$$right\_name? : NAME$$
$$\overline{}$$
$$(task?, right\_name?) \in port\_set\_namep$$

$$NotC2MachPortGetSetStatusNameIsAPortSet$$
$$\mathrel{\widehat=} PortNameSpace \land \lnot\ C2MachPortGetSetStatusNameIsAPortSet$$

8.5.4 Return Values

Table 6 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The order of the checks agrees with the code in CM on 14Sep94.

Review Note:
This request can also return $Kern\_resource\_shortage$, when allocating memory for $member\_names$.

In the code from which this model was produced, one page of memory is allocated for $member\_names$ before checking either C1 or C2. If C1 and C2 are both true, then an exhaustive search of $task?$'s name space is performed to search for receive rights to any of the ports in the port set named by $right\_name?$. If more rights are found than will fit on one page of memory, then additional memory is allocated at this time.

Thus $Kern\_resource\_shortage$ may be returned before or after checking C1 and C2. If it is returned after the other checks, $member\_names$ will contain a partial list of names.

$return!$	$member\_names!$	$count!$	C1	C2
$Kern\_invalid\_name$	-	-	F	-
$Kern\_invalid\_right$	-	-	T	F
$Kern\_success$	as described below	as described below	T	T

Table 6: Return Values for **mach_port_get_set_status**

$$
\begin{array}{l}
\textit{RVMachPortGetSetStatusInvalidName} \\
\hline
MachPortGetSetStatusOutputs \\
NotC1MachPortGetSetStatusNameIsARight \\
\hline
return! = Kern\_invalid\_name
\end{array}
$$

$$
\begin{array}{l}
\textit{RVMachPortGetSetStatusInvalidRight} \\
\hline
MachPortGetSetStatusOutputs \\
C1MachPortGetSetStatusNameIsARight \\
NotC2MachPortGetSetStatusNameIsAPortSet \\
\hline
return! = Kern\_invalid\_right
\end{array}
$$

$$
\begin{array}{l}
\textit{RVMachPortGetSetStatusSuccess} \\
\hline
MachPortGetSetStatusOutputs \\
C1MachPortGetSetStatusNameIsARight \\
C2MachPortGetSetStatusNameIsAPortSet \\
\hline
return! = Kern\_success \\
member\_names! = \{port : PORT \\
\quad \mid port \in port\_set(task?, right\_name?) \bullet receiver\_name(port)\} \\
count! = \#member\_names!
\end{array}
$$

8.5.5 State Changes

A **mach_port_get_set_status** request does not make any state changes since it only observes the system state.

8.5.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_get_set_status** request is described in Section 8.1.

$$\begin{array}{|l} \hline \textit{ProcessingMachPortGetSetStatus} \\ \textit{ProcessPortRequestGood} \\ \hline \textit{operation}? = \textit{Mach\_port\_get\_set\_status\_id} \\ \hline \end{array}$$

An unsuccessful **mach_port_get_set_status** request results in no changes to the Mach state and returns only the appropriate error status.

$$\begin{aligned} &\textit{MachPortGetSetStatusBad} \\ &\quad \widehat{=} (\textit{RVMachPortGetSetStatusInvalidName} \\ &\qquad \lor \textit{RVMachPortGetSetStatusInvalidRight}) \\ &\quad \gg \textit{RequestNoOp} \end{aligned}$$

A successful **mach_port_get_set_status** request results in no changes to the Mach state and returns a reply message.

$$\begin{aligned} &\textit{MachPortGetSetStatusGood} \\ &\quad \widehat{=} \textit{RVMachPortGetSetStatusSuccess} \\ &\quad \gg \textit{MachPortGetSetStatusReply} \end{aligned}$$

The complete specification of kernel processing of a **mach_port_get_set_status** request consists of the initial processing followed by an unsuccessful or successful execution.

$$\begin{aligned} &\textit{MachPortGetSetStatus} \\ &\quad \widehat{=} \textit{ProcessingMachPortGetSetStatus} \\ &\quad ⨾ (\textit{MachPortGetSetStatusBad} \\ &\qquad \lor \textit{MachPortGetSetStatusGood}) \end{aligned}$$

8.6 mach_port_names

A **mach_port_names** request returns information about all of the rights in a task's port name space. The same information for a single right can be retrieved using **mach_port_type**.

8.6.1 Client Interface

kern_return_t **mach_port_names**
(mach_port_t *task_name,*

mach_port_array_t*	*right_names*,
mach_msg_type_number_t*	*ncount*,
mach_port_type_array_t*	*type_masks*,
mach_msg_type_number_t*	*tcount*);

8.6.1.1 Input Parameters The following input parameters are provided by the client of a **mach_port_names** request:

- $task\_name?$ — the client's name for the task whose port name space is returned

$MachPortNamesClientInputs$
$task\_name? : NAME$

A **mach_port_names** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_names\_id$ and an empty body.

$InvokeMachPortNames$
$InvokeMachMsg$
$MachPortNamesClientInputs$

$name? = task\_name?$
$operation? = Mach\_port\_names\_id$

8.6.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_names** request:

- $return!$ — the status of the request
- $right\_names!$ — a sequence consisting of all names in the port name space of $task\_name?$
- $ncount!$ — the number of elements in the sequence $right\_names!$
- $type\_masks!$ — a sequence of port type masks, as described in Section 8.1.4, corresponding to each element of the sequence $right\_names!$
- $tcount!$ — the number of elements in the sequence $type\_masks!$ (which is the same as $ncount!$)

$MachPortNamesClientOutputs$
$return! : KERNEL\_RETURN$
$right\_names! : \mathrm{seq}\ NAME$
$ncount! : \mathbb{N}$
$type\_masks! : \mathrm{seq}\ PortTypeMask$
$tcount! : \mathbb{N}$

$MachPortNamesReceiveReply$
$InvokeMachMsgRcv$
$MachPortNamesClientOutputs$

$(return!, right\_names!, ncount!, type\_masks!, tcount!)$
$= Body\_to\_mach\_port\_names\_outputs(msg\_body)$

8.6.2 Kernel Interface

8.6.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_names** request:

- $task?$ — the task known to the client by $task\_name?$

$$\begin{array}{|l} \hline \textit{MachPortNamesInputs} \\ task? : TASK \\ \hline \end{array}$$

8.6.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_names** request:

- $return!$ — the status of the request
- $right\_names!$ — a sequence consisting of all names in the port name space of $task\_name?$
- $ncount!$ — the number of elements in the sequence $right\_names!$
- $type\_masks!$ — a sequence of port type masks, as described in Section 8.1.4, corresponding to each element of the sequence $right\_names!$
- $tcount!$ — the number of elements in the sequence $type\_masks!$ (which is the same as $ncount!$)

$$\begin{array}{|l} \hline \textit{MachPortNamesOutputs} \\ return! : KERNEL\_RETURN \\ right\_names! : \mathrm{seq}\ NAME \\ ncount! : \mathbb{N} \\ type\_masks! : \mathrm{seq}\ PortTypeMask \\ tcount! : \mathbb{N} \\ \hline \end{array}$$

Upon completion of the processing of a **mach_port_names** request, a reply message is built from the output parameters.

$$\begin{array}{|l} \hline \textit{MachPortNamesReply} \\ RequestOnlyObserves \\ right\_names? : \mathrm{seq}\ NAME \\ ncount? : \mathbb{N} \\ type\_masks? : \mathrm{seq}\ PortTypeMask \\ tcount? : \mathbb{N} \\ \hline reply? \\ = Mach\_port\_names\_outputs\_to\_reply(right\_names?, ncount?, type\_masks?, tcount?) \\ \hline \end{array}$$

8.6.3 Request Criteria

There are no criteria for the **mach_port_names** request.

8.6.4 Return Values

Table 7 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$	$right\_names!$	$ncount!$	$type\_masks!$	$tcount!$
$Kern\_success$	described below	$\# \, right\_names!$	described below	$\# \, type\_masks!$

Table 7: Return Values for **mach_port_names**

Review Note:
There is also the possibility that $Kern\_resource\_shortage$ can be returned, though this has not been modeled.

Review Note:
No specification is given for the $mach\_port\_type\_msg\_accepted\_request$ field, since the model currently does not include the necessary information.

Review Note:
There is actually also a COMPAT field in the return mask. However, in the prototype it will always return false so it need not be modeled.

$right\_names!$ is a sequence consisting of all names in the name space for $task?$. No element of the name space is repeated, so the number of elements in $right\_names!$ is $number\_of\_rights(task?)$.

$ncount!$ is the number of elements in $right\_names!$.

$type\_masks!$ is a sequence, whose length $tcount!$ is the same as $ncount!$, consisting of a $PortTypeMask$ for each corresponding name in $right\_names!$. Each mask contains boolean flags indicating the following:

- $mach\_port\_type\_send$— if the name refers to a send right
- $mach\_port\_type\_receive$— if the name refers to a receive right
- $mach\_port\_type\_send\_once$— if the name refers to a send-once right
- $mach\_port\_type\_port\_set$— if the name refers to a port set
- $mach\_port\_type\_dead\_name$— if the name refers to a dead right
- $mach\_port\_type\_dead\_name\_request$— if there is an outstanding dead-name notification request for the name

$$
\begin{array}{l}
\textit{RVMachPortNamesSuccess} \\
\hline
\textit{MachPortNamesOutputs} \\
\textit{PortNameSpace} \\
\textit{Notifications} \\
task? : TASK \\
\hline
return! = Kern\_success \\
\mathrm{ran}\ right\_names! = local\_namep ⦇\{task?\}⦈ \\
ncount! = \#right\_names! = number\_of\_rights(task?) \\
tcount! = \#type\_masks! = ncount! \\
\forall i : 1 \mathinner{\ldotp\ldotp} ncount! \\
\bullet (((type\_masks!(i)).mach\_port\_type\_send = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in s\_right) \\
\land ((type\_masks!(i)).mach\_port\_type\_receive = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in r\_right) \\
\land ((type\_masks!(i)).mach\_port\_type\_send\_once = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in so\_right) \\
\land ((type\_masks!(i)).mach\_port\_type\_port\_set = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in port\_set\_namep) \\
\land ((type\_masks!(i)).mach\_port\_type\_dead\_name = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in dead\_namep) \\
\land ((type\_masks!(i)).mach\_port\_type\_dead\_name\_request = True \\
\quad \Leftrightarrow (task?, right\_names!(i)) \in \mathrm{dom}\ port\_notify\_dead)) \\
\hline
\end{array}
$$

8.6.5 State Changes

A **mach_port_names** request does not make any state changes since it only observes the system state.

8.6.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_names** request is described in Section 8.1.

$$
\begin{array}{l}
\textit{ProcessingMachPortNames} \\
\hline
\textit{ProcessPortRequestGood} \\
\hline
operation? = Mach\_port\_names\_id \\
\hline
\end{array}
$$

A **mach_port_names** request results in no changes to the Mach state and returns a reply message.

$$
\begin{array}{l}
\textit{MachPortNamesGood} \\
\quad \widehat{=}\ \textit{RVMachPortNamesSuccess} \\
\quad \gg \textit{MachPortNamesReply}
\end{array}
$$

The complete specification of kernel processing of a **mach_port_names** request consists of the initial processing followed by execution.

$$MachPortNames \\ \quad \widehat{=} \ ProcessingMachPortNames \\ \quad \fatsemi \ MachPortNamesGood$$

8.7 mach_port_rename

A **mach_port_rename** request allows a client to change the name by which a task knows a port, port set or dead name.

8.7.1 Client Interface

```
kern_return_t mach_port_rename
                (mach_port_t            task_name,
                 mach_port_t            old_name,
                 mach_port_t            new_name);
```

8.7.1.1 Input Parameters The following input parameters are provided by the client of a **mach_port_rename** request:

- $task\_name?$ — the client's name for the task whose port name space is to be changed
- $old\_name?$ — the current name of the right to be renamed
- $new\_name?$ — the new name for the right

$$\begin{array}{l} \text{MachPortRenameClientInputs} \\ \hline task\_name? : NAME \\ old\_name? : NAME \\ new\_name? : NAME \\ \hline \end{array}$$

A **mach_port_rename** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_rename\_id$ and has a body consisting of $old\_name?$ and $new\_name?$.

$$\begin{array}{l} \text{InvokeMachPortRename} \\ \hline InvokeMachMsg \\ MachPortRenameClientInputs \\ \hline name? = task\_name? \\ operation? = Mach\_port\_rename\_id \\ msg\_body = Mach\_port\_rename\_inputs\_to\_body(old\_name?, new\_name?) \\ \hline \end{array}$$

8.7.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **mach_port_rename** request:

- $return!$ — the status of the request

$$\begin{array}{|l}\hline MachPortRename\,ClientOutputs \\ return! : KERNEL\_RETURN \\ \hline\end{array}$$

$$\begin{array}{|l}\hline MachPortRename\,ReceiveReply \\ Invoke\,MachMsgRcv \\ MachPortRename\,ClientOutputs \\ \hline return! = Body\_to\_mach\_port\_rename\_outputs(msg\_body) \\ \hline\end{array}$$

8.7.2 Kernel Interface

8.7.2.1 Input Parameters

The following input parameters are provided to the kernel for a **mach_port_rename** request:

- $task?$ — the task known to the client by $task\_name?$
- $old\_name?$ — the current name of the right to be renamed
- $new\_name?$ — the new name for the right

$$\begin{array}{|l}\hline MachPortRename\,Inputs \\ task? : TASK \\ old\_name? : NAME \\ new\_name? : NAME \\ \hline\end{array}$$

8.7.2.2 Output Parameters

The following output parameters are returned by the kernel for a **mach_port_rename** request:

- $return!$ — the status of the request

$$\begin{array}{|l}\hline MachPortRename\,Outputs \\ return! : KERNEL\_RETURN \\ \hline\end{array}$$

8.7.3 Request Criteria

The following criteria are defined for the **mach_port_rename** request:

- **C1** — $new\_name?$ is not a reserved name.

$$\begin{array}{|l}\hline C1MachPortRenameNewNameNotReserved \\ new\_name? : NAME \\ \hline new\_name? \notin Reserved\_names \\ \hline\end{array}$$

$$NotC1MachPortRenameNewNameNotReserved \\ \quad \widehat{=} \neg\ C1MachPortRenameNewNameNotReserved$$

- **C2** — $new\_name?$ is not currently in the name space of $task?$.

$$\begin{array}{l} C2MachPortRenameNewNameNotInUse \\ \hline PortNameSpace \\ task? : TASK \\ new\_name? : NAME \\ \hline (task?, new\_name?) \notin local\_namep \end{array}$$

$$NotC2MachPortRenameNewNameNotInUse \\ \quad \widehat{=} PortNameSpace \wedge \neg\ C2MachPortRenameNewNameNotInUse$$

- **C3** — $old\_name?$ is currently in the name space of $task?$.

$$\begin{array}{l} C3MachPortRenameOldNameInUse \\ \hline PortNameSpace \\ task? : TASK \\ old\_name? : NAME \\ \hline (task?, old\_name?) \in local\_namep \end{array}$$

$$NotC3MachPortRenameOldNameInUse \\ \quad \widehat{=} PortNameSpace \wedge \neg\ C3MachPortRenameOldNameInUse$$

8.7.4 Return Values

Table 8 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The criteria are in the correct order according to the code in CM on 16Sep94.

Review Note:
This request can also return $Kern\_resource\_shortage$, because it may need to create a new entry. This check, if it occurs, comes between C1 and C2.

$$\begin{array}{l} RVMachPortRenameInvalidValue \\ \hline MachPortRenameOutputs \\ NotC1MachPortRenameNewNameNotReserved \\ \hline return! = Kern\_invalid\_value \end{array}$$

$return!$	C1	C2	C3
$Kern\_invalid\_value$	F	-	-
$Kern\_name\_exists$	T	F	-
$Kern\_invalid\_name$	T	T	F
$Kern\_success$	T	T	T

Table 8: Return Values for **mach_port_rename**

$RVMachPortRenameNameExists$
$MachPortRename\,Outputs$
$C1MachPortRenameNewNameNotReserved$
$NotC2MachPortRenameNewNameNotInUse$

$return! = Kern\_name\_exists$

$RVMachPortRenameInvalidName$
$MachPortRename\,Outputs$
$C1MachPortRenameNewNameNotReserved$
$C2MachPortRenameNewNameNotInUse$
$NotC3MachPortRenameOldNameInUse$

$return! = Kern\_invalid\_name$

$RVMachPortRenameSuccess$
$MachPortRename\,Outputs$
$C1MachPortRenameNewNameNotReserved$
$C2MachPortRenameNewNameNotInUse$
$C3MachPortRenameOldNameInUse$

$return! = Kern\_success$

8.7.5 State Changes

If all of the criteria are satisfied, then the name space for $task?$ is changed so that the name of any right currently with the name $old\_name?$ is changed to $new\_name?$. Here right refers to a send, receive, or send once right, port set or dead name.

In addition, if there is an outstanding dead-name request for the right $old\_name?$, then the request must be renamed.

Review Note:
In addition, the name of a message accepted request may need to be changed. However, that is not currently in the model.

There are several things in the model that should be changed by this request, but are not because the model is incorrect.

- A name in a port set. The model is incorrect, since the kernel actually considers port sets as a set of ports, not a set of names.
- A name in the set of registered rights ($registered\_rights$) for a task. Again, the model is incorrect since this is really a set of ports, not a set of names. (This is modeled correctly in the FSPM.)

- The data structures associated with threads blocked for a pending receive ($PendingReceive$ and $\underline{p}ending\_receives$). Again, the model is incorrect in considering a port to be a name.

$$
\begin{array}{l}
\textit{MachPortRenameState} \\
\Delta\ PortNameSpace \\
\Delta\ Notifications \\
C1MachPortRenameNewNameNotReserved \\
C2MachPortRenameNewNameNotInUse \\
C3MachPortRenameOldNameInUse \\
\hline
\underline{p}ort\_right\_rel' = \underline{p}ort\_right\_rel \\
\quad \setminus \{ port : PORT;\ right : RIGHT;\ i : \mathbb{N}_1 \bullet (task?, port, old\_name?, right, i)\} \\
\quad \cup \{ port : PORT;\ right : RIGHT;\ i : \mathbb{N}_1 \\
\qquad \mid (task?, port, old\_name?, right, i) \in \underline{p}ort\_right\_rel \\
\qquad \bullet (task?, port, new\_name?, right, i)\} \\
\underline{p}ort\_set\_rel' = \underline{p}ort\_set\_rel \\
\quad \setminus \{set\_of\_ports : \mathbb{P}\ PORT \bullet (task?, old\_name?, set\_of\_ports)\} \\
\quad \cup \{set\_of\_ports : \mathbb{P}\ PORT \\
\qquad \mid (task?, old\_name?, set\_of\_ports) \in \underline{p}ort\_set\_rel \\
\qquad \bullet (task?, new\_name?, set\_of\_ports)\} \\
\underline{d}ead\_right\_rel' = \underline{d}ead\_right\_rel \\
\quad \setminus \{ i : \mathbb{N}_1 \bullet (task?, old\_name?, i)\} \\
\quad \cup \{ i : \mathbb{N}_1 \mid (task?, old\_name?, i) \in \underline{d}ead\_right\_rel \\
\qquad \bullet (task?, new\_name?, i)\} \\
\underline{p}ort\_notify\_dead\_rel' = \underline{p}ort\_notify\_dead\_rel \\
\quad \setminus \{ port : PORT \bullet (port, task?, old\_name?)\} \\
\quad \cup \{ port : PORT \mid (port, task?, old\_name?) \in \underline{p}ort\_notify\_dead\_rel \\
\qquad \bullet (port, task?, new\_name?)\}
\end{array}
$$

Review Note:
Invariants should be stated here as well.

8.7.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_rename** request is described in Section 8.1.

$$
\begin{array}{l}
\textit{ProcessingMachPortRename} \\
ProcessPortRequestGood \\
\hline
operation? = Mach\_port\_rename\_id
\end{array}
$$

An unsuccessful **mach_port_rename** request results in no changes to the Mach state and returns only the appropriate error status.

$$
\begin{array}{l}
MachPortRenameBad \\
\quad \mathrel{\widehat=} (RVMachPortRenameNameExists \\
\quad\quad \lor RVMachPortRenameInvalidName \\
\quad\quad \lor RVMachPortRenameInvalidValue) \\
\quad \gg RequestNoOp
\end{array}
$$

A successful **mach_port_rename** request alters the Mach state as described in Section 8.7.5 and returns a reply message.

$$
\begin{array}{l}
MachPortRenameGood \\
\quad \mathrel{\widehat=} (MachPortRenameState \\
\quad \land RVMachPortRenameSuccess) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

The complete specification of kernel processing of a **mach_port_rename** request consists of the initial processing followed by an unsuccessful or successful execution.

$$
\begin{array}{l}
MachPortRename \\
\quad \mathrel{\widehat=} ProcessingMachPortRename \\
\quad \fatsemi (MachPortRenameBad \\
\quad\quad \lor MachPortRenameGood)
\end{array}
$$

8.8 mach_port_request_notification

A **mach_port_request_notification** request registers a notification message to be sent when a particular port event occurs. If a notification has already been requested, it returns the send-once right associated with the existing notification request.

Notifications are described in Section 8.1.7.

8.8.1 Client Interface

```
kern_return_t mach_port_request_notification
              (mach_port_t                task_name,
               mach_port_t                right_name,
               mach_msg_id_t              variant,
               mach_port_mscount_t        sync,
               mach_port_t                notify_name,
               mach_msg_type_name_t       notify_type,
               mach_port_t*               previous_name);
```

8.8.1.1 Input Parameters

The following input parameters are provided by the client of a **mach_port_request_notification** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located

- $right\_name?$ — the name of a right, in $task\_name?$'s name space, for which the notification is requested. If $variant?$ is set to $Mach\_notify\_port\_destroyed$ or $Mach\_notify\_no\_senders$, this must be a receive right.
- $variant?$ — the type of event for which notification is requested, either $Mach\_notify\_port\_destroyed$, $Mach\_notify\_no\_senders$, or $Mach\_notify\_dead\_name$
- $sync?$ — When $variant?$ is set to $Mach\_notify\_dead\_name$, this must be set to zero. When $variant?$ is set to $Mach\_notify\_no\_senders$, this value is used to overcome race conditions
- $notify\_name?$ — the name of a right, in the client's name space, for the port to which the notification should be send
- $notify\_type?$ — the manner in which a send-once right should be extracted from $notify\_name?$, either $Mmt\_make\_send\_once$ or $Mmt\_move\_send\_once$

$MachPortRequestNotificationClientInputs$

$$
\begin{array}{l}
task\_name? : NAME \\
right\_name? : NAME \\
variant? : MACH\_MSG\_ID \\
sync? : \mathbb{N} \\
notify\_name? : NAME \\
notify\_type? : MACH\_MSG\_TYPE
\end{array}
$$

A **mach_port_request_notification** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_request\_notification\_id$ and has a body consisting of $right\_name?$, $variant?$, $sync?$, $notify\_name?$, and $notify\_type?$.

$InvokeMachPortRequestNotification$

$$
\begin{array}{l}
InvokeMachMsg \\
MachPortRequestNotificationClientInputs \\
\hline
name? = task\_name? \\
operation? = Mach\_port\_request\_notification\_id \\
msg\_body = Mach\_port\_request\_notification\_inputs\_to\_body(right\_name?, variant?, \\
\quad sync?, notify\_name?, notify\_type?)
\end{array}
$$

8.8.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **mach_port_request_notification** request:

- $return!$ — the status of the request
- $previous\_name!$ — if the notification has already been requested, the previously registered send-once right; otherwise $Mach\_port\_null$

$MachPortRequestNotificationClientOutputs$

$$
\begin{array}{l}
return! : KERNEL\_RETURN \\
previous\_name! : NAME
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{__ MachPortRequestNotificationReceiveReply ____} \\
InvokeMachMsgRcv \\
MachPortRequestNotificationClientOutputs \\
\hline
(return!, previous_name!) = Body_to_mach_port_request_notification_outputs(msg_body) \\
\hline
\end{array}$$

8.8.2 Kernel Interface

8.8.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_request_notification** request:

- $task?$ — the task known to the client by $task\_name?$
- $right\_name?$ — provided by the client
- $variant?$ — provided by the client
- $sync?$ — provided by the client
- $notify?$ — the port known to the client by $notify\_name?$

$$\begin{array}{|l}
\multicolumn{1}{l}{__ MachPortRequestNotificationInputs ____} \\
task? : TASK \\
right_name? : NAME \\
variant? : MACH_MSG_ID \\
sync? : \mathbb{N} \\
notify? : PORT \\
\hline
\end{array}$$

8.8.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_request_notification** request:

- $return!$ — the status of the request
- $previous!$ — if the notification has already been requested, the previously registered notification port; otherwise Ip_null

$$\begin{array}{|l}
\multicolumn{1}{l}{__ MachPortRequestNotificationOutputs ____} \\
return! : KERNEL_RETURN \\
previous! : PORT \\
\hline
\end{array}$$

Upon completion of the processing of a **mach_port_request_notification** request, a reply message is built from the output parameters.

$$\begin{array}{|l}
\multicolumn{1}{l}{__ MachPortRequestNotificationReply ____} \\
RequestReturn \\
previous? : PORT \\
\hline
reply? = Mach_port_request_notification_outputs_to_reply(previous?) \\
\hline
\end{array}$$

8.8.3 Request Criteria

The following criteria are defined for the **mach_port_request_notification** request:

Review Note:
There is a number missing in the list because of improvements to the model. It doesn't seem to be worthwhile to renumber the criteria.

- **C1** — $notify?$ is not Ip_dead.

$$\begin{array}{l} C1MachPortRequestNotificationNotifyNotDead \\ \hline notify? : PORT \\ \hline notify? \neq Ip\_dead \end{array}$$

$$NotC1MachPortRequestNotificationNotifyNotDead \\ \quad \widehat{=} \neg\ C1MachPortRequestNotificationNotifyNotDead$$

- **C2** — $variant?$ is set to $Mach\_notify\_port\_destroyed$.

$$\begin{array}{l} C2MachPortRequestNotificationPortDestroyed \\ \hline variant? : MACH\_MSG\_ID \\ \hline variant? = Mach\_notify\_port\_destroyed \end{array}$$

$$NotC2MachPortRequestNotificationPortDestroyed \\ \quad \widehat{=} \neg\ C2MachPortRequestNotificationPortDestroyed$$

- **C3** — $variant?$ is set to $Mach\_notify\_no\_senders$.

$$\begin{array}{l} C3MachPortRequestNotificationNoSenders \\ \hline variant? : MACH\_MSG\_ID \\ \hline variant? = Mach\_notify\_no\_senders \end{array}$$

$$NotC3MachPortRequestNotificationNoSenders \\ \quad \widehat{=} \neg\ C3MachPortRequestNotificationNoSenders$$

- **C4** — $variant?$ is set to $Mach\_notify\_dead\_name$.

$$\begin{array}{l} C4MachPortRequestNotificationDeadName \\ \hline variant? : MACH\_MSG\_ID \\ \hline variant? = Mach\_notify\_dead\_name \end{array}$$

$$NotC4MachPortRequestNotificationDeadName \\ \quad \widehat{=} \neg\ C4MachPortRequestNotificationDeadName$$

- **C5** — $sync?$ is equal to zero.

$$\begin{array}{|l}\hline C5MachPortRequestNotificationSyncIsZero \\ \hline sync? : \mathbb{N} \\ \hline sync? = 0 \\ \hline \end{array}$$

$NotC5MachPortRequestNotificationSyncIsZero$
$\quad \widehat{=} \neg\ C5MachPortRequestNotificationSyncIsZero$

- **C6** — $right\_name?$ represents a right in $task?$'s name space.

$$\begin{array}{|l}\hline C6MachPortRequestNotificationNameIsARight \\ \hline PortNameSpace \\ task? : TASK \\ right\_name? : NAME \\ \hline (task?, right\_name?) \in local\_namep \\ \hline \end{array}$$

$NotC6MachPortRequestNotificationNameIsARight$
$\quad \widehat{=} PortNameSpace \land \neg\ C6MachPortRequestNotificationNameIsARight$

- **C7** — $right\_name?$ represents a receive right in $task?$'s name space.

$$\begin{array}{|l}\hline C7MachPortRequestNotificationNameIsAReceiveRight \\ \hline PortNameSpace \\ task? : TASK \\ right\_name? : NAME \\ \hline (task?, right\_name?) \in r\_right \\ \hline \end{array}$$

$NotC7MachPortRequestNotificationNameIsAReceiveRight$
$\quad \widehat{=} PortNameSpace \land \neg\ C7MachPortRequestNotificationNameIsAReceiveRight$

- **C8** — This criteria only applies if $right\_name?$ indicates a receive right in $task?$'s name space, in which case it is true whenever:
  - There are no send rights for the port indicated by $right\_name?$,
  - $sync?$ is less than the make-send count value for that port, and
  - $notify?$ is not Ip_null.

Review Note:
This criteria is not completely defined in the schema because the state model does not capture the total number of send rights associated with a port.

$$\begin{array}{|l}\hline C8MachPortRequestNoSendersNotificationSendNow \\ \hline C7MachPortRequestNotificationNameIsAReceiveRight \\ SendRightsCount \\ sync? : \mathbb{N} \\ notify? : PORT \\ \hline sync? \leq \underline{m}ake\_send\_count(named\_port(task?, right\_name?)) \\ notify? \neq Ip\_null \\ \hline \end{array}$$

$NotC8MachPortRequestNoSendersNotificationSendNow$
$\widehat{=}$ $C7MachPortRequestNotificationNameIsAReceiveRight \land SendRightsCount$
$\land \neg\ C8MachPortRequestNoSendersNotificationSendNow$

- **C10** — $right\_name?$ represents a send, send-once or receive right in $task?$'s name space.

$C10MachPortRequestNotificationNameIsAPortRight$
$PortNameSpace$
$task? : TASK$
$right\_name? : NAME$

$(task?, right\_name?) \in port\_right\_namep$

$NotC10MachPortRequestNotificationNameIsAPortRight$
$\widehat{=}$ $PortNameSpace \land \neg\ C10MachPortRequestNotificationNameIsAPortRight$

- **C11** — $right\_name?$ represents a deadname in $task?$'s name space.

$C11MachPortRequestNotificationNameIsADeadName$
$PortNameSpace$
$task? : TASK$
$right\_name? : NAME$

$(task?, right\_name?) \in dead\_namep$

$NotC11MachPortRequestNotificationNameIsADeadName$
$\widehat{=}$ $PortNameSpace \land \neg\ C11MachPortRequestNotificationNameIsADeadName$

- **C12** — $notify?$ is not Ip_null and $sync?$ is non-zero.

$C12MachPortRequestNotificationNotifySyncNotZero$
$notify? : PORT$
$sync? : \mathbb{N}$

$notify? \neq Ip\_null$
$sync? \neq 0$

$NotC12MachPortRequestNotificationNotifyAndSyncNonZero$
$\widehat{=}$ $\neg\ C12MachPortRequestNotificationNotifySyncNotZero$

- **C13** — The number of user references for $right\_name?$ is less than the maximum allowed.

$C13MachPortRequestNotificationURefsNotAtMax$
$PortNameSpace$
$task? : TASK$
$right\_name? : NAME$

$dead\_right\_ref\_count(task?, right\_name?) < Max\_right\_refs$

$NotC13MachPortRequestNotificationURefsNotAtMax$
$\widehat{=}$ $PortNameSpace \land \neg\ C13MachPortRequestNotificationURefsNotAtMax$

8.8.4 Return Values

Table 9 describes those values returned at the completion of the request which are common to all three values of $variant?$. Tables 10, 11 and 12 describe the return values specific to the cases in which $variant?$ is set to $Mach\_notify\_port\_destroyed$, $Mach\_notify\_no\_senders$, and $Mach\_notify\_dead\_name$, respectively.

Review Note:
The order of the checks in this section is accurate based upon the code in CM on 19Sep94.

Review Note:
In Table 9, note that C2, C3, and C4 are mutually exclusive.

$return!$	$previous!$	C1	C2	C3	C4
$Kern\_invalid\_capability$	-	F	-	-	-
$Kern\_invalid\_value$	-	T	F	F	F
See Table 10		T	T	-	-
See Table 11		T	-	T	-
See Table 12		T	-	-	T

Table 9: Return Values for **mach_port_request_notification**

$RVMachPortRequestNotificationInvalidCapability$
$MachPortRequestNotification\ Outputs$
$NotC1MachPortRequestNotificationNotifyNotDead$

$return! = Kern\_invalid\_capability$

$RVMachPortRequestNotificationInvalidValue$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$NotC2MachPortRequestNotificationPortDestroyed$
$NotC3MachPortRequestNotificationNoSenders$
$NotC4MachPortRequestNotificationDeadName$

$return! = Kern\_invalid\_value$

8.8.4.1 Port-Destroyed Notification Request

Table 10 describes the return values for the case in which $variant?$ is set to $Mach\_notify\_port\_destroyed$.

$RVMachPortRequestPortDestroyedNotificationInvalidValue$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C2MachPortRequestNotificationPortDestroyed$
$NotC5MachPortRequestNotificationSyncIsZero$

$return! = Kern\_invalid\_value$

$return!$	$previous!$	C1	C2	C5	C6	C7
$Kern\_invalid\_value$	-	T	T	F	-	-
$Kern\_invalid\_name$	-	T	T	T	F	-
$Kern\_invalid\_right$	-	T	T	T	T	F
$Kern\_success$	$port\_notify\_destroyed($ $named\_port(task?,right\_name?))$	T	T	T	T	T

Table 10: Return Values for **mach_port_request_notification**, port-destroyed notification

$RVMachPortRequestPortDestroyedNotificationInvalidName$
$MachPortRequestNotificationOutputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C2MachPortRequestNotificationPortDestroyed$
$C5MachPortRequestNotificationSyncIsZero$
$NotC6MachPortRequestNotificationNameIsARight$

$return! = Kern\_invalid\_name$

$RVMachPortRequestPortDestroyedNotificationInvalidRight$
$MachPortRequestNotificationOutputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C2MachPortRequestNotificationPortDestroyed$
$C5MachPortRequestNotificationSyncIsZero$
$C6MachPortRequestNotificationNameIsARight$
$NotC7MachPortRequestNotificationNameIsAReceiveRight$

$return! = Kern\_invalid\_right$

$RVMachPortRequestPortDestroyedNotificationSuccess$
$Notifications$
$MachPortRequestNotificationOutputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C2MachPortRequestNotificationPortDestroyed$
$C5MachPortRequestNotificationSyncIsZero$
$C6MachPortRequestNotificationNameIsARight$
$C7MachPortRequestNotificationNameIsAReceiveRight$

$return! = Kern\_success$
$previous! = port\_notify\_destroyed(named\_port(task?, right\_name?))$

8.8.4.2 No-Senders Notification Request

Table 11 describes the return values for the case in which $variant?$ is set to $Mach\_notify\_no\_senders$.

$RVMachPortRequestNoSendersNotificationInvalidName$
$MachPortRequestNotificationOutputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C3MachPortRequestNotificationNoSenders$
$NotC6MachPortRequestNotificationNameIsARight$

$return! = Kern\_invalid\_name$

$return!$	$previous!$	C1	C3	C6	C7
$Kern\_invalid\_name$	-	T	T	F	-
$Kern\_invalid\_right$	-	T	T	T	F
$Kern\_success$	$port\_notify\_no\_more\_senders($ $named\_port(task?, right\_name?))$	T	T	T	T

Table 11: Return Values for **mach_port_request_notification**, no-senders notification

$RVMachPortRequestNoSendersNotificationInvalidRight$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C3MachPortRequestNotificationNoSenders$
$C6MachPortRequestNotificationNameIsARight$
$NotC7MachPortRequestNotificationNameIsAReceiveRight$

$return! = Kern\_invalid\_right$

$RVMachPortRequestNoSendersNotificationSuccess$
$Notifications$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C3MachPortRequestNotificationNoSenders$
$C6MachPortRequestNotificationNameIsARight$
$C7MachPortRequestNotificationNameIsAReceiveRight$

$return! = Kern\_success$
$previous! = port\_notify\_no\_more\_senders(named\_port(task?, right\_name?))$

8.8.4.3 Dead-Name Notification Request

Table 12 describes the return values for the case in which $variant?$ is set to $Mach\_notify\_dead\_name$.

Review Note:
In Table 12, note that C10 and C11 are mutually exclusive.

$return!$	$previous!$	C1	C4	C6	C10	C11	C12	C13
$Kern\_invalid\_name$	-	T	T	F	-	-	-	-
$Kern\_invalid\_right$	-	T	T	T	F	F	-	-
$Kern\_success$	$port\_notify\_dead($ $task?, right\_name?)$	T	T	T	T	-	-	-
$Kern\_invalid\_argument$	-	T	T	T	-	T	F	-
$Kern\_urefs\_overflow$	-	T	T	T	-	T	T	F
$Kern\_success$	Ip_null	T	T	T	-	T	T	T

Table 12: Return Values for **mach_port_request_notification**, dead-name notification

$RVMachPortRequestDeadNameNotificationInvalidName$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C4MachPortRequestNotificationDeadName$
$NotC6MachPortRequestNotificationNameIsARight$

$return! = Kern\_invalid\_name$

$RVMachPortRequestDeadNameNotificationInvalidRight$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C4MachPortRequestNotificationDeadName$
$C6MachPortRequestNotificationNameIsARight$
$NotC10MachPortRequestNotificationNameIsAPortRight$
$NotC11MachPortRequestNotificationNameIsADeadName$

$return! = Kern\_invalid\_right$

$RVMachPortRequestDeadNameNotificationSuccessOne$
$Notifications$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C4MachPortRequestNotificationDeadName$
$C6MachPortRequestNotificationNameIsARight$
$C10MachPortRequestNotificationNameIsAPortRight$

$return! = Kern\_success$
$previous! = port\_notify\_dead(task?, right\_name?)$

Review Note:
Note that this case could also result in a resource shortage. This is due to the fact that many dead name notifications can be active for the same port, so memory is allocated for each additional notification request.

$RVMachPortRequestDeadNameNotificationInvalidArgument$
$MachPortRequestNotification\ Outputs$
$C1MachPortRequestNotificationNotifyNotDead$
$C4MachPortRequestNotificationDeadName$
$C6MachPortRequestNotificationNameIsARight$
$C11MachPortRequestNotificationNameIsADeadName$
$NotC12MachPortRequestNotificationNotifyAndSyncNonZero$

$return! = Kern\_invalid\_argument$

Editorial Note:
It seems rather surprising that the request fails when $name?$ is a dead name and $notify?$ is $Ip\_null?$ That seems like a case that Mach would usually allow, though the request in this case would do nothing.

RVMachPortRequestDeadNameNotificationUrefsOverflow

MachPortRequestNotification Outputs
C1MachPortRequestNotificationNotifyNotDead
C4MachPortRequestNotificationDeadName
C6MachPortRequestNotificationNameIsARight
C11MachPortRequestNotificationNameIsADeadName
C12MachPortRequestNotificationNotifySyncNotZero
NotC13MachPortRequestNotificationURefsNotAtMax
$return! = Kern\_urefs\_overflow$

RVMachPortRequestDeadNameNotificationSuccessTwo

MachPortRequestNotification Outputs
C1MachPortRequestNotificationNotifyNotDead
C4MachPortRequestNotificationDeadName
C6MachPortRequestNotificationNameIsARight
C11MachPortRequestNotificationNameIsADeadName
C12MachPortRequestNotificationNotifySyncNotZero
C13MachPortRequestNotificationURefsNotAtMax
$return! = Kern\_success$
$previous! = Ip\_null$

In this last case, $previous!$ returns Ip_null since $right\_name?$ identifies a dead right.

8.8.5 State Changes

Table 13 lists the possible successful executions of a **mach_port_request_notification** request.

Case	C1	C2	C3	C4	C5	C6	C7	C8	C10	C11	C12	C13
MachPortRequestPortDestroyedNotificationStateChanges	T	T	-	-	T	T	T	-	-	-	-	-
MachPortRequestNoSendersNotificationStateChangesOne	T	-	T	-	-	T	T	F	-	-	-	-
MachPortRequestNoSendersNotificationStateChangesTwo	T	-	T	-	-	T	T	T	-	-	-	-
MachPortRequestDeadNameNotificationStateChangesOne	T	-	-	T	-	T	-	-	T	-	-	-
MachPortRequestDeadNameNotificationStateChangesTwo	T	-	-	T	-	T	-	-	-	T	T	T

Table 13: State Change Cases for **mach_port_request_notification**

8.8.5.1 Port-Destroyed Notification Request

A request for a port-destroyed notification is successful if the following are true:

- $notify?$ is not Ip_dead
- $variant?$ is set to $Mach\_notify\_port\_destroyed$
- $sync?$ is zero
- $right\_name?$ is a receive right in $task?$'s name space

If the request is successful, the kernel replaces the existing port-destroyed notification request port for the port named by $right\_name?$ with $notify?$. Note that either the existing notification request port or $notify?$ could be Ip_null.

$$
\begin{array}{l}
\textit{MachPortRequestPortDestroyedNotificationStateChanges} \\
\hline
\Delta \; Notifications \\
C1MachPortRequestNotificationNotifyNotDead \\
C2MachPortRequestNotificationPortDestroyed \\
C5MachPortRequestNotificationSyncIsZero \\
C6MachPortRequestNotificationNameIsARight \\
C7MachPortRequestNotificationNameIsAReceiveRight \\
\hline
port\_notify\_destroyed' = port\_notify\_destroyed \\
\quad \oplus \{named\_port(task?, right\_name?) \mapsto notify?\}
\end{array}
$$

8.8.5.2 No-Senders Notification Request A request for a no-senders notification is successful if the following are true:

- $notify?$ is not Ip_dead
- $variant?$ is set to $Mach\_notify\_no\_senders$
- $right\_name?$ is a receive right in $task?$'s name space

If the request is successful, then kernel further determines whether a no-senders notification should immediately be sent, by checking if all of the following are true:

- There are no send rights for the port indicated by $right\_name?$
- $sync?$ is less than the make-send count value for that port
- $notify?$ is not Ip_null

If any of these are not true, the kernel replaces the existing no-senders notification request port for the port named by $right\_name?$ with $notify?$. Note that either the existing notification request port or $notify?$ could be Ip_null.

$$
\begin{array}{l}
\textit{MachPortRequestNoSendersNotificationStateChangesOne} \\
\hline
\Delta \; Notifications \\
C1MachPortRequestNotificationNotifyNotDead \\
C3MachPortRequestNotificationNoSenders \\
C6MachPortRequestNotificationNameIsARight \\
C7MachPortRequestNotificationNameIsAReceiveRight \\
NotC8MachPortRequestNoSendersNotificationSendNow \\
\hline
port\_notify\_no\_more\_senders' = port\_notify\_no\_more\_senders \\
\quad \oplus \{named\_port(task?, right\_name?) \mapsto notify?\}
\end{array}
$$

If the kernel determines that it must immediately send a no-senders notification, it first removes any existing notification request port and then attempts to send the notification.

$$
\begin{array}{|l}
\hline
\textit{MachPortRequestNoSendersNotificationStateChangesTwo} \\
\Delta\ Notifications \\
C1MachPortRequestNotificationNotifyNotDead \\
C3MachPortRequestNotificationNoSenders \\
C6MachPortRequestNotificationNameIsARight \\
C7MachPortRequestNotificationNameIsAReceiveRight \\
C8MachPortRequestNoSendersNotificationSendNow \\
\hline
port\_notify\_no\_more\_senders' = port\_notify\_no\_more\_senders \\
\quad \oplus \{named\_port(task?, right\_name?) \mapsto Ip\_null\} \\
\hline
\end{array}
$$

Review Note:
This second case still needs to be completed. Four things are missing, all of which should eventually be handled in the port chapter introduction:

- Make sure that a notification message can be allocated. If not, no notification is sent.
- Build the notification message.
- Check for send permission
- Queue the message

Review Note:
It's interesting to note that the $sync?$ value is used to determine whether a no-senders notification is immediately sent, but it has no relevance to notifications sent after the notification is registered. In that respect it seems that the race condition that $sync?$ is apparently intended to prevent is not really prevented. The only way to truly avoid it is for the recipient of the notification to check the make send count returned with the notification.

$$
\begin{array}{l}
MachPortRequestNoSendersNotificationStateChanges \\
\quad \widehat{=}\ MachPortRequestNoSendersNotificationStateChangesOne \\
\qquad \lor\ MachPortRequestNoSendersNotificationStateChangesTwo
\end{array}
$$

8.8.5.3 Dead-Name Notification Request

A request for a dead-name notification can be successful in two distinct cases.

The first successful case occurs if the following are true:

- $notify?$ is not Ip_dead
- $variant?$ is set to $Mach\_notify\_dead\_name$
- $right\_name?$ is a send, send-once or receive right in $task?$'s name space

In this case, the kernel replaces the existing dead-name notification request port for $right\_name?$ with $notify?$. Note that either the existing notification request port or $notify?$ could be Ip_null.

$$
\begin{array}{|l}
\hline
\textit{MachPortRequestDeadNameNotificationStateChangesOne} \\
\Delta\ Notifications \\
C1MachPortRequestNotificationNotifyNotDead \\
C4MachPortRequestNotificationDeadName \\
C6MachPortRequestNotificationNameIsARight \\
C10MachPortRequestNotificationNameIsAPortRight \\
\hline
port\_notify\_dead' = port\_notify\_dead \oplus \{(task?, right\_name?) \mapsto notify?\} \\
\hline
\end{array}
$$

The second successful case occurs if the following are true:

- $notify?$ is not Ip_dead
- $variant?$ is set to $Mach\_notify\_dead\_name$
- $right\_name?$ is a dead right in $task?$'s name space
- $sync?$ is non-zero
- $dead\_right\_ref\_count(task?, right\_name?)$ is not at the maximum value (Max_right_refs)

In this case, the kernel immediately attempts to send a dead-name notification. Prior to doing this however, it increments the $dead\_right\_ref\_count$ for the dead right.

$$
\begin{array}{l}
\textit{MachPortRequestDeadNameNotificationStateChangesTwo} \\
\hline
\Delta\ DeadRights \\
C1MachPortRequestNotificationNotifyNotDead \\
C4MachPortRequestNotificationDeadName \\
C6MachPortRequestNotificationNameIsARight \\
C11MachPortRequestNotificationNameIsADeadName \\
C12MachPortRequestNotificationNotifySyncNotZero \\
C13MachPortRequestNotificationURefsNotAtMax \\
\hline
dead\_right\_ref\_count' = dead\_right\_ref\_count \\
\quad \oplus \{(task?, right\_name?) \mapsto dead\_right\_ref\_count(task?, right\_name?) + 1\} \\
\hline
\end{array}
$$

Editorial Note:
It is unclear why $dead\_right\_ref\_count$ is being incremented in this case. That does not agree with the understanding of this field presented in the state model. It might also be interesting to look and see whether it is decremented in the case that the send of the notification message fails.

Review Note:
This second case still needs to be completed. Four things are missing, all of which should eventually be handled in the port chapter introduction:

- Make sure that a notification message can be allocated. If not, no notification is sent.
- Build the notification message.
- Check for send permission
- Queue the message

$$
\begin{array}{l}
MachPortRequestDeadNameNotificationStateChanges \\
\quad \widehat{=}\ MachPortRequestDeadNameNotificationStateChangesOne \\
\qquad \lor\ MachPortRequestDeadNameNotificationStateChangesTwo
\end{array}
$$

8.8.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_request_notification** request is described in Section 8.1.

$$
\begin{array}{l}
\textit{ProcessingMachPortRequestNotification} \\
\hline
ProcessPortRequestGood \\
\hline
operation? = Mach\_port\_request\_notification\_id \\
\hline
\end{array}
$$

An unsuccessful **mach_port_request_notification** request results in no changes to the Mach state and returns only the appropriate error status.

$$
\begin{aligned}
& MachPortRequestNotificationBad \\
& \quad \widehat{=} (RVMachPortRequestNotificationInvalidCapability \\
& \qquad \lor RVMachPortRequestNotificationInvalidValue \\
& \qquad \lor RVMachPortRequestPortDestroyedNotificationInvalidValue \\
& \qquad \lor RVMachPortRequestPortDestroyedNotificationInvalidName \\
& \qquad \lor RVMachPortRequestPortDestroyedNotificationInvalidRight \\
& \qquad \lor RVMachPortRequestNoSendersNotificationInvalidName \\
& \qquad \lor RVMachPortRequestNoSendersNotificationInvalidRight \\
& \qquad \lor RVMachPortRequestDeadNameNotificationInvalidName \\
& \qquad \lor RVMachPortRequestDeadNameNotificationInvalidRight \\
& \qquad \lor RVMachPortRequestDeadNameNotificationInvalidArgument \\
& \qquad \lor RVMachPortRequestDeadNameNotificationUrefsOverflow) \\
& \quad \gg RequestNoOp
\end{aligned}
$$

A successful **mach_port_request_notification** request alters the Mach state as described in Section 8.8.5 and returns a reply message.

$$
\begin{aligned}
& MachPortRequestNotificationGood \\
& \quad \widehat{=} ((MachPortRequestPortDestroyedNotificationStateChanges \\
& \qquad \lor MachPortRequestNoSendersNotificationStateChanges \\
& \qquad \lor MachPortRequestDeadNameNotificationStateChanges) \\
& \quad \land (RVMachPortRequestPortDestroyedNotificationSuccess \\
& \qquad \lor RVMachPortRequestNoSendersNotificationSuccess \\
& \qquad \lor RVMachPortRequestDeadNameNotificationSuccessOne \\
& \qquad \lor RVMachPortRequestDeadNameNotificationSuccessTwo)) \\
& \quad \gg MachPortRequestNotificationReply
\end{aligned}
$$

The complete specification of kernel processing of a **mach_port_request_notification** request consists of the initial processing followed by an unsuccessful or successful execution.

$$
\begin{aligned}
& MachPortRequestNotification \\
& \quad \widehat{=} ProcessingMachPortRequestNotification \\
& \quad \fatsemi (MachPortRequestNotificationBad \\
& \qquad \lor MachPortRequestNotificationGood)
\end{aligned}
$$

8.9 mach_port_set_mscount

A **mach_port_set_mscount** request changes the make-send count for the port associated with a specified receive right in a task's port name space.

8.9.1 Client Interface

```
kern_return_t mach_port_set_mscount
              (mach_port_t              task_name,
               mach_port_t              right_name,
               mach_port_mscount_t      mscount);
```

8.9.1.1 Input Parameters

The following input parameters are provided by the client of a **mach_port_set_mscount** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of a receive right for the port whose make-send count is to be changed
- $mscount?$ — the value to be assigned to the make-send count for the port associated with $right\_name?$

MachPortSetMscountClientInputs

$$task\_name? : NAME$$
$$right\_name? : NAME$$
$$mscount? : \mathbb{N}$$

A **mach_port_set_mscount** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_set\_mscount\_id$ and has a body consisting of $right\_name?$ and $mscount?$.

InvokeMachPortSetMscount

$$InvokeMachMsg$$
$$MachPortSetMscountClientInputs$$

$$name? = task\_name?$$
$$operation? = Mach\_port\_set\_mscount\_id$$
$$msg\_body = Mach\_port\_set\_mscount\_inputs\_to\_body(right\_name?, mscount?)$$

8.9.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **mach_port_set_mscount** request:

- $return!$ — the status of the request

MachPortSetMscountClientOutputs

$$return! : KERNEL\_RETURN$$

MachPortSetMscountReceiveReply

$$InvokeMachMsgRcv$$
$$MachPortSetMscountClientOutputs$$

$$return! = Body\_to\_mach\_port\_set\_mscount\_outputs(msg\_body)$$

8.9.2 Kernel Interface

8.9.2.1 Input Parameters

The following input parameters are provided to the kernel for a **mach_port_set_mscount** request:

- $task?$ — the task known to the client by $task\_name?$

- $right\_name?$ — provided by the client
- $mscount?$ — provided by the client

$$
\begin{array}{|l}
\hline
\textit{MachPortSetMscountInputs} \\
task? : TASK \\
right\_name? : NAME \\
mscount? : \mathbb{N} \\
\hline
\end{array}
$$

8.9.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_set_mscount** request:

- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{MachPortSetMscountOutputs} \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

8.9.3 Request Criteria

The following criteria are defined for the **mach_port_set_mscount** request:

- **C1** — $right\_name?$ represents a right in $task?$'s name space.

$$
\begin{array}{|l}
\hline
\textit{C1MachPortSetMscountNameIsARight} \\
PortNameSpace \\
task? : TASK \\
right\_name? : NAME \\
\hline
(task?, right\_name?) \in local\_namep \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1MachPortSetMscountNameIsARight \\
\quad \widehat{=} \ PortNameSpace \land \neg \ C1MachPortSetMscountNameIsARight
\end{array}
$$

- **C2** — $right\_name?$ represents a receive right in $task?$'s name space.

$$
\begin{array}{|l}
\hline
\textit{C2MachPortSetMscountNameIsAReceiveRight} \\
PortNameSpace \\
task? : TASK \\
right\_name? : NAME \\
\hline
(task?, right\_name?) \in r\_right \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC2MachPortSetMscountNameIsAReceiveRight \\
\quad \widehat{=} \ PortNameSpace \land \neg \ C2MachPortSetMscountNameIsAReceiveRight
\end{array}
$$

8.9.4 Return Values

Table 14 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The order of the checks agrees with the code in CM as of 20Sep94.

$return!$	C1	C2
$Kern\_invalid\_name$	F	-
$Kern\_invalid\_right$	T	F
$Kern\_success$	T	T

Table 14: Return Values for **mach_port_set_mscount**

$RVMachPortSetMscountInvalidName$
$MachPortSetMscountOutputs$
$NotC1MachPortSetMscountNameIsARight$

$return! = Kern\_invalid\_name$

$RVMachPortSetMscountInvalidRight$
$MachPortSetMscountOutputs$
$C1MachPortSetMscountNameIsARight$
$NotC2MachPortSetMscountNameIsAReceiveRight$

$return! = Kern\_invalid\_right$

$RVMachPortSetMscountSuccess$
$MachPortSetMscountOutputs$
$C1MachPortSetMscountNameIsARight$
$C2MachPortSetMscountNameIsAReceiveRight$

$return! = Kern\_success$

8.9.5 State Changes

If all of the criteria are satisfied, then the make-send count for the port associated with $right\_name?$ in $task?$'s name space is given the value $mscount?$.

$MachPortSetMscountState$
$\Delta\ PortSummary$
$MachPortSetMscountInputs$
$C1MachPortSetMscountNameIsARight$
$C2MachPortSetMscountNameIsAReceiveRight$

$\underline{m}ake\_send\_count'$
$\quad = \underline{m}ake\_send\_count \oplus \{named\_port(task?, right\_name?) \mapsto mscount?\}$

Review Note:
Invariants should be stated here as well.

8.9.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_set_mscount** request is described in Section 8.1.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ ProcessingMachPortSetMscount\ \_\_\_\_\_\_\_\_} \\
ProcessPortRequestGood \\
\hline
operation? = Mach\_port\_set\_mscount\_id \\
\hline
\end{array}
$$

An unsuccessful **mach_port_set_mscount** request results in no changes to the Mach state and returns only the appropriate error status.

$$
\begin{array}{l}
MachPortSetMscountBad \\
\quad \widehat{=} (RVMachPortSetMscountInvalidName \\
\quad\quad \lor RVMachPortSetMscountInvalidRight) \\
\quad \gg RequestNoOp
\end{array}
$$

A successful **mach_port_set_mscount** request alters the Mach state as described in Section 8.9.5 and returns a reply message.

$$
\begin{array}{l}
MachPortSetMscountGood \\
\quad \widehat{=} (MachPortSetMscountState \\
\quad \land RVMachPortSetMscountSuccess) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

The complete specification of kernel processing of a **mach_port_set_mscount** request consists of the initial processing followed by an unsuccessful or successful execution.

$$
\begin{array}{l}
MachPortSetMscount \\
\quad \widehat{=} ProcessingMachPortSetMscount \\
\quad \fatsemi (MachPortSetMscountBad \\
\quad\quad \lor MachPortSetMscountGood)
\end{array}
$$

8.10 mach_port_set_qlimit

A **mach_port_set_qlimit** request changes the message queue limit for the port associated with a specified receive right in a task's port name space.

8.10.1 Client Interface

kern_return_t **mach_port_set_qlimit**
(mach_port_t *task_name*,
mach_port_t *rightname*,
mach_port_msgcount_t *qlimit_value*);

8.10.1.1 Input Parameters The following input parameters are provided by the client of a **mach_port_set_qlimit** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of a receive right for the port whose message queue limit is to be changed
- $qlimit\_value?$ — the value to be assigned to the message queue limit for the port associated with $right\_name?$

$$\begin{array}{l} \textit{MachPortSetQlimitClientInputs} \\ \hline task\_name? : NAME \\ right\_name? : NAME \\ qlimit\_value? : \mathbb{N} \end{array}$$

A **mach_port_set_qlimit** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_set\_qlimit\_id$ and has a body consisting of $right\_name?$ and $qlimit\_value?$.

$$\begin{array}{l} \textit{InvokeMachPortSetQlimit} \\ \hline InvokeMachMsg \\ MachPortSetQlimitClientInputs \\ \hline name? = task\_name? \\ operation? = Mach\_port\_set\_qlimit\_id \\ msg\_body = Mach\_port\_set\_qlimit\_inputs\_to\_body(right\_name?, qlimit\_value?) \end{array}$$

8.10.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **mach_port_set_qlimit** request:

- $return!$ — the status of the request

$$\begin{array}{l} \textit{MachPortSetQlimitClientOutputs} \\ \hline return! : KERNEL\_RETURN \end{array}$$

$$\begin{array}{l} \textit{MachPortSetQlimitReceiveReply} \\ \hline InvokeMachMsgRcv \\ MachPortSetQlimitClientOutputs \\ \hline return! = Body\_to\_mach\_port\_set\_qlimit\_outputs(msg\_body) \end{array}$$

8.10.2 Kernel Interface

8.10.2.1 Input Parameters The following input parameters are provided to the kernel for a **mach_port_set_qlimit** request:

- $task?$ — the task known to the client by $task\_name?$

- $right\_name?$ — provided by the client
- $qlimit\_value?$ — provided by the client

$$
\begin{array}{l}
\textit{MachPortSetQlimitInputs} \\
\hline
task? : TASK \\
right\_name? : NAME \\
qlimit\_value? : \mathbb{N} \\
\hline
\end{array}
$$

8.10.2.2 Output Parameters The following output parameters are returned by the kernel for a **mach_port_set_qlimit** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{MachPortSetQlimitOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

8.10.3 Request Criteria

The following criteria are defined for the **mach_port_set_qlimit** request:

- **C1** — $qlimit\_value?$ is no larger than the specified maximum, $Mach\_port\_q\_limit\_max$.

$$
\begin{array}{l}
\textit{C1MachPortSetQlimitNameIsARight} \\
\hline
qlimit\_value? : \mathbb{N} \\
\hline
qlimit\_value? \leq Mach\_port\_q\_limit\_max \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1MachPortSetQlimitNameIsARight \\
\quad \widehat{=} \neg\ C1MachPortSetQlimitNameIsARight
\end{array}
$$

- **C2** — $right\_name?$ represents a right in $task?$'s name space.

$$
\begin{array}{l}
\textit{C2MachPortSetQlimitNameIsAReceiveRight} \\
\hline
PortNameSpace \\
task? : TASK \\
right\_name? : NAME \\
\hline
(task?, right\_name?) \in local\_namep \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC2MachPortSetQlimitNameIsAReceiveRight \\
\quad \widehat{=}\ PortNameSpace \wedge \neg\ C2MachPortSetQlimitNameIsAReceiveRight
\end{array}
$$

- **C3** — $right\_name?$ represents a receive right in $task?$'s name space.

$C3MachPortSetQlimitValueIsValid$
$PortNameSpace$
$task? : TASK$
$right\_name? : NAME$

$(task?, right\_name?) \in r\_right$

$$NotC3MachPortSetQlimitValueIsValid \mathrel{\widehat=} PortNameSpace \land \lnot C3MachPortSetQlimitValueIsValid$$

8.10.4 Return Values

Table 15 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The order of the checks agrees with the code in CM as of 20Sep94.

$return!$	C1	C2	C3
$Kern\_invalid\_value$	F	-	-
$Kern\_invalid\_name$	T	F	-
$Kern\_invalid\_right$	T	T	F
$Kern\_success$	T	T	T

Table 15: Return Values for **mach_port_set_qlimit**

$RVMachPortSetQlimitInvalidValue$
$MachPortSetQlimitOutputs$
$NotC1MachPortSetQlimitNameIsARight$

$return! = Kern\_invalid\_value$

$RVMachPortSetQlimitInvalidName$
$MachPortSetQlimitOutputs$
$C1MachPortSetQlimitNameIsARight$
$NotC2MachPortSetQlimitNameIsAReceiveRight$

$return! = Kern\_invalid\_name$

$RVMachPortSetQlimitInvalidRight$
$MachPortSetQlimitOutputs$
$C1MachPortSetQlimitNameIsARight$
$C2MachPortSetQlimitNameIsAReceiveRight$
$NotC3MachPortSetQlimitValueIsValid$

$return! = Kern\_invalid\_right$

$$\begin{array}{|l}
\hline
\textit{RVMachPortSetQlimitSuccess} \\
\textit{MachPortSetQlimitOutputs} \\
\textit{C1MachPortSetQlimitNameIsARight} \\
\textit{C2MachPortSetQlimitNameIsAReceiveRight} \\
\textit{C3MachPortSetQlimitValueIsValid} \\
\hline
return! = Kern_success \\
\hline
\end{array}$$

8.10.5 State Changes

If all of the criteria are satisfied, then the message queue limit for the port associated with $right\_name?$ in $task?$'s name space is given the value $qlimit\_value?$.

Review Note:
This request may wake up threads which are blocked trying to send to the port, if the queue limit is increased. This does not currently fit into the model.

$$\begin{array}{|l}
\hline
\textit{MachPortSetQlimitState} \\
\Delta\ PortSummary \\
\textit{C1MachPortSetQlimitNameIsARight} \\
\textit{C2MachPortSetQlimitNameIsAReceiveRight} \\
\textit{C3MachPortSetQlimitValueIsValid} \\
\hline
\underline{q}_limit' = \underline{q}_limit \oplus \{named_port(task?, right_name?) \mapsto qlimit_value?\} \\
\hline
\end{array}$$

Review Note:
Invariants should be stated here as well.

8.10.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_set_qlimit** request is described in Section 8.1.

$$\begin{array}{|l}
\hline
\textit{ProcessingMachPortSetQlimit} \\
\textit{ProcessPortRequestGood} \\
\hline
operation? = Mach_port_set_qlimit_id \\
\hline
\end{array}$$

An unsuccessful **mach_port_set_qlimit** request results in no changes to the Mach state and returns only the appropriate error status.

$$\begin{array}{l}
MachPortSetQlimitBad \\
\quad \widehat{=} (RVMachPortSetQlimitInvalidName \\
\qquad \lor RVMachPortSetQlimitInvalidRight \\
\qquad \lor RVMachPortSetQlimitInvalidValue) \\
\quad \gg RequestNoOp
\end{array}$$

A successful **mach_port_set_qlimit** request alters the Mach state as described in Section 8.10.5 and returns a reply message.

$$\begin{aligned}&MachPortSetQlimitGood\\&\quad \mathrel{\widehat=} (MachPortSetQlimitState\\&\quad \land RVMachPortSetQlimitSuccess)\\&\quad \gg RequestReturnOnlyStatus\end{aligned}$$

The complete specification of kernel processing of a **mach_port_set_qlimit** request consists of the initial processing followed by an unsuccessful or successful execution.

$$\begin{aligned}&MachPortSetQlimit\\&\quad \mathrel{\widehat=} ProcessingMachPortSetQlimit\\&\quad \mathrel{;} (MachPortSetQlimitBad\\&\qquad \lor MachPortSetQlimitGood)\end{aligned}$$

8.11 mach_port_set_seqno

A **mach_port_set_seqno** request changes the current sequence number for the port associated with a specified receive right in a task's port name space.

8.11.1 Client Interface

```
kern_return_t mach_port_set_seqno
              (mach_port_t          task_name,
               mach_port_t          right_name,
               mach_port_seqno_t    seqno);
```

8.11.1.1 Input Parameters

The following input parameters are provided by the client of a **mach_port_set_seqno** request:

- $task\_name?$ — the client's name for the task in whose name space $right\_name?$ is located
- $right\_name?$ — the name of a receive right for the port whose current sequence number is to be changed
- $seqno?$ — the value to be assigned to the current sequence number for the port associated with $right\_name?$

$$\begin{array}{|l}\hline MachPortSetSeqnoClientInputs\\ task\_name? : NAME\\ right\_name? : NAME\\ seqno? : \mathbb{Z}\\ \hline\end{array}$$

A **mach_port_set_seqno** request is invoked by sending a message to the port indicated by $task\_name?$ that has the operation field set to $Mach\_port\_set\_seqno\_id$ and has a body consisting of $right\_name?$ and $seqno?$.

$$\begin{array}{l}
\textit{InvokeMachPortSetSeqno} \\
\hline
\textit{InvokeMachMsg} \\
\textit{MachPortSetSeqnoClientInputs} \\
\hline
name? = task_name? \\
operation? = Mach_port_set_seqno_id \\
msg_body = Mach_port_set_seqno_inputs_to_body(right_name?, seqno?)
\end{array}$$

8.11.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **mach_port_set_seqno** request:

- $return!$ — the status of the request

$$\begin{array}{l}
\textit{MachPortSetSeqnoClientOutputs} \\
\hline
return! : KERNEL_RETURN
\end{array}$$

$$\begin{array}{l}
\textit{MachPortSetSeqnoReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{MachPortSetSeqnoClientOutputs} \\
\hline
return! = Body_to_mach_port_set_seqno_outputs(msg_body)
\end{array}$$

8.11.2 Kernel Interface

8.11.2.1 Input Parameters

The following input parameters are provided to the kernel for a **mach_port_set_seqno** request:

- $task?$ — the task known to the client by $task\_name?$
- $right\_name?$ — provided by the client
- $seqno?$ — provided by the client

$$\begin{array}{l}
\textit{MachPortSetSeqnoInputs} \\
\hline
task? : TASK \\
right_name? : NAME \\
seqno? : \mathbb{Z}
\end{array}$$

8.11.2.2 Output Parameters

The following output parameters are returned by the kernel for a **mach_port_set_seqno** request:

- $return!$ — the status of the request

$$\begin{array}{l}
\textit{MachPortSetSeqnoOutputs} \\
\hline
return! : KERNEL_RETURN
\end{array}$$

8.11.3 Request Criteria

The following criteria are defined for the **mach_port_set_seqno** request:

- **C1** — $right\_name?$ represents a right in $task?$'s name space.

$$
\begin{array}{|l}
\hline
C1MachPortSetSeqnoNameIsARight \\
PortNameSpace \\
task? : TASK \\
right\_name? : NAME \\
\hline
(task?, right\_name?) \in local\_namep \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1MachPortSetSeqnoNameIsARight \\
\quad \widehat{=} \; PortNameSpace \land \lnot \; C1MachPortSetSeqnoNameIsARight
\end{array}
$$

- **C2** — $right\_name?$ represents a receive right in $task?$'s name space.

$$
\begin{array}{|l}
\hline
C2MachPortSetSeqnoNameIsAReceiveRight \\
PortNameSpace \\
task? : TASK \\
right\_name? : NAME \\
\hline
(task?, right\_name?) \in r\_right \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC2MachPortSetSeqnoNameIsAReceiveRight \\
\quad \widehat{=} \; PortNameSpace \land \lnot \; C2MachPortSetSeqnoNameIsAReceiveRight
\end{array}
$$

8.11.4 Return Values

Table 16 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
The order of the checks agrees with the code in CM as of 20Sep94.

$return!$	C1	C2
$Kern\_invalid\_name$	F	-
$Kern\_invalid\_right$	T	F
$Kern\_success$	T	T

Table 16: Return Values for **mach_port_set_seqno**

$$
\begin{array}{|l}
\hline
RVMachPortSetSeqnoInvalidName \\
MachPortSetSeqnoOutputs \\
NotC1MachPortSetSeqnoNameIsARight \\
\hline
return! = Kern\_invalid\_name \\
\hline
\end{array}
$$

$$\begin{array}{|l} \hline RVMachPortSetSeqnoInvalidRight \\ MachPortSetSeqnoOutputs \\ C1MachPortSetSeqnoNameIsARight \\ NotC2MachPortSetSeqnoNameIsAReceiveRight \\ \hline return! = Kern\_invalid\_right \\ \hline \end{array}$$

$$\begin{array}{|l} \hline RVMachPortSetSeqnoSuccess \\ MachPortSetSeqnoOutputs \\ C1MachPortSetSeqnoNameIsARight \\ C2MachPortSetSeqnoNameIsAReceiveRight \\ \hline return! = Kern\_success \\ \hline \end{array}$$

8.11.5 State Changes

If all of the criteria are satisfied, then the current sequence number for the port associated with $right\_name?$ in $task?$'s name space is given the value $seqno?$.

$$\begin{array}{|l} \hline MachPortSetSeqnoState \\ \Delta\ PortSummary \\ MachPortSetSeqnoInputs \\ C1MachPortSetSeqnoNameIsARight \\ C2MachPortSetSeqnoNameIsAReceiveRight \\ \hline \underline{s}equence\_no' = \underline{s}equence\_no \oplus \{named\_port(task?, right\_name?) \mapsto seqno?\} \\ \hline \end{array}$$

Review Note:
Invariants should be stated here as well.

8.11.6 Complete Request

The initial processing by the kernel upon receipt of the **mach_port_set_seqno** request is described in Section 8.1.

$$\begin{array}{|l} \hline ProcessingMachPortSetSeqno \\ ProcessPortRequestGood \\ \hline operation? = Mach\_port\_set\_seqno\_id \\ \hline \end{array}$$

An unsuccessful **mach_port_set_seqno** request results in no changes to the Mach state and returns only the appropriate error status.

$$\begin{array}{l} MachPortSetSeqnoBad \\ \quad \widehat{=} (RVMachPortSetSeqnoInvalidName \\ \qquad \vee RVMachPortSetSeqnoInvalidRight) \\ \quad \gg RequestNoOp \end{array}$$

A successful **mach_port_set_seqno** request alters the Mach state as described in Section 8.11.5 and returns a reply message.

$$
\begin{aligned}
&MachPortSetSeqnoGood \\
&\quad \widehat{=} \ (MachPortSetSeqnoState \\
&\quad \land \ RVMachPortSetSeqnoSuccess) \\
&\quad \gg \ RequestReturnOnlyStatus
\end{aligned}
$$

The complete specification of kernel processing of a **mach_port_set_seqno** request consists of the initial processing followed by an unsuccessful or successful execution.

$$
\begin{aligned}
&MachPortSetSeqno \\
&\quad \widehat{=} \ ProcessingMachPortSetSeqno \\
&\quad \fatsemi \ (MachPortSetSeqnoBad \\
&\qquad \lor \ MachPortSetSeqnoGood)
\end{aligned}
$$

Section 9

Thread Requests

9.1 Introduction to Thread Requests

This chapter describes the thread kernel requests in DTOS.

9.1.1 Constants and Types

We first define the identifier that is used to represent each thread request. The kernel accepts two thread requests through task kernel ports ($Thread\_task\_port\_ops$) and most of the others through thread kernel ports ($Thread\_thread\_port\_ops$).

$$
\begin{array}{l}
Thread\_abort\_id, Thread\_assign\_id, Thread\_assign\_default\_id,\\
Thread\_depress\_abort\_id, Thread\_disable\_pc\_sampling\_id,\\
Thread\_enable\_pc\_sampling\_id, Thread\_get\_assignment\_id,\\
Thread\_get\_sampled\_pcs\_id, Thread\_get\_special\_port\_id,\\
Thread\_get\_state\_id, Thread\_info\_id, Thread\_max\_priority\_id,\\
Thread\_policy\_id, Thread\_priority\_id, Thread\_resume\_id,\\
Thread\_resume\_secure\_id, Thread\_set\_special\_port\_id,\\
Thread\_set\_state\_id, Thread\_set\_state\_secure\_id,\\
Thread\_suspend\_id, Thread\_terminate\_id : OPERATION\\
Thread\_thread\_port\_ops : \mathbb{P}\ OPERATION\\
\hline
\langle Thread\_abort\_id, Thread\_assign\_id, Thread\_assign\_default\_id,\\
\quad Thread\_depress\_abort\_id, Thread\_disable\_pc\_sampling\_id,\\
\quad Thread\_enable\_pc\_sampling\_id, Thread\_get\_assignment\_id,\\
\quad Thread\_get\_sampled\_pcs\_id, Thread\_get\_special\_port\_id,\\
\quad Thread\_get\_state\_id, Thread\_info\_id, Thread\_max\_priority\_id,\\
\quad Thread\_policy\_id, Thread\_priority\_id, Thread\_resume\_id,\\
\quad Thread\_resume\_secure\_id, Thread\_set\_special\_port\_id,\\
\quad Thread\_set\_state\_id, Thread\_set\_state\_secure\_id,\\
\quad Thread\_suspend\_id, Thread\_terminate\_id\rangle\\
\qquad Values\_partition\ Thread\_thread\_port\_ops
\end{array}
$$

$$
\begin{array}{l}
Thread\_create\_id, Thread\_create\_secure\_id : OPERATION\\
Thread\_task\_port\_ops : \mathbb{P}\ OPERATION\\
\hline
\langle Thread\_create\_id, Thread\_create\_secure\_id\rangle\\
\quad Values\_partition\ Thread\_task\_port\_ops
\end{array}
$$

Together these two disjoint sets of operations form the set $Thread\_operations$ denoting all thread operations. Each thread request must be received through a port of the appropriate port class.

$$\begin{array}{|l} Thread\_operations : \mathbb{P}\ OPERATION \\ \hline \langle Thread\_thread\_port\_ops, Thread\_task\_port\_ops \rangle \text{ partition } Thread\_operations \\ Thread\_thread\_port\_ops \subseteq Allowed\_mach\_services(Pc\_thread) \\ Thread\_task\_port\_ops \subseteq Allowed\_mach\_services(Pc\_task) \end{array}$$

9.1.2 Required Permissions

For each operation there is a primary permission that is required to perform the operation. We define here the portion of the $Required\_permission$ function that pertains to thread requests. The $Abort\_thread$ implementation service permission implies the $Set\_thread\_priority$ and $Abort\_thread\_depress$ permissions are automatically granted since the **thread_abort** request can set priorities and abort priority depression. The **thread_priority** request requires $Set\_thread\_priority$ permission, but $Set\_max\_thread\_priority$ permission is also needed if the set_max parameter has value $True$. We also assume that $Initiate\_secure$ permission is granted whenever $Resume\_thread$ or Set_thread_state permission is granted.

Review Note:
Here are the full sets of permissions that are currently needed for each request (except the special port ones).

$$\begin{array}{l}
\{(Thread_abort_id, Abort_thread, Set_thread_priority, Abort_thread_depress), \\
(Thread_assign_id, Assign_thread_to_pset, Set_max_thread_priority, \\
\quad Set_thread_priority, Set_thread_policy, Assign_thread), \\
(Thread_assign_default_id, Assign_thread_to_pset, Set_max_thread_priority, \\
\quad Set_thread_priority, Set_thread_policy, Assign_thread), \\
(Thread_create_id, Add_thread), \\
(Thread_depress_abort_id, Abort_thread_depress, Set_thread_priority), \\
(Thread_disable_pc_sampling_id, Sample_thread), \\
(Thread_enable_pc_sampling_id, Sample_thread), \\
(Thread_get_assignment_id, Get_thread_assignment), \\
(Thread_get_sampled_pcs_id, Sample_thread), \\
(Thread_get_state_id, Get_thread_state), \\
(Thread_info_id, Get_thread_info), \\
(Thread_max_priority_id, Set_max_thread_priority, Set_thread_priority), \\
(Thread_policy_id, Set_thread_policy), \\
(Thread_priority_id, Set_thread_priority, Set_max_thread_priority), \\
(Thread_resume_id, Resume_thread, Initiate_secure), \\
(Thread_set_state_id, Set_thread_state, Initiate_secure), \\
(Thread_suspend_id, Suspend_thread), \\
(Thread_terminate_id, Terminate_thread, Sample_thread)\}
\end{array}$$

This will be simplified when the FSPM is modified so services do not overlap so often.

Review Note:
Does **thread_assign_default** also require $Assign\_thread$ permission? I suspect so.

$$
\begin{aligned}
&\{(Thread\_abort\_id, Abort\_thread),\\
&\quad (Thread\_assign\_id, Assign\_thread\_to\_pset),\\
&\quad (Thread\_assign\_default\_id, Assign\_thread\_to\_pset),\\
&\quad (Thread\_create\_id, Add\_thread),\\
&\quad (Thread\_create\_secure\_id, Add\_thread\_secure),\\
&\quad (Thread\_depress\_abort\_id, Abort\_thread\_depress),\\
&\quad (Thread\_disable\_pc\_sampling\_id, Sample\_thread),\\
&\quad (Thread\_enable\_pc\_sampling\_id, Sample\_thread),\\
&\quad (Thread\_get\_assignment\_id, Get\_thread\_assignment),\\
&\quad (Thread\_get\_sampled\_pcs\_id, Sample\_thread),\\
&\quad (Thread\_get\_state\_id, Get\_thread\_state),\\
&\quad (Thread\_info\_id, Get\_thread\_info),\\
&\quad (Thread\_max\_priority\_id, Set\_max\_thread\_priority),\\
&\quad (Thread\_policy\_id, Set\_thread\_policy),\\
&\quad (Thread\_priority\_id, Set\_thread\_priority),\\
&\quad (Thread\_resume\_id, Resume\_thread),\\
&\quad (Thread\_resume\_secure\_id, Initiate\_secure),\\
&\quad (Thread\_set\_state\_id, Set\_thread\_state),\\
&\quad (Thread\_set\_state\_secure\_id, Initiate\_secure),\\
&\quad (Thread\_suspend\_id, Suspend\_thread),\\
&\quad (Thread\_terminate\_id, Terminate\_thread)\}\\
&\subset Required\_permission
\end{aligned}
$$

The permission required for a $Thread\_get\_special\_port\_id$ or $Thread\_set\_special\_port\_id$ operation depends upon the value of the $which\_port$? parameter. Therefore the permission cannot be checked in the common processing, and the two operations are in the set$Service_check_deferred$.

$$\{Thread\_get\_special\_port\_id, Thread\_set\_special\_port\_id\} \subseteq Service\_check\_deferred$$

9.1.3 Invariant Information

The thread requests operate on only certain components of the state. We use the following schema to provide a general framework for describing thread requests.

Review Note:
This list has problems in that some schemas are indirectly pulled in where they should not be. For example, $\Xi\ SpecialTaskPorts$ includes $\Xi\ PortExist$ which we do not want. Might be possible to get a better handle on this problem by doing fuzz -t and comparing $\Delta\ DtosExec$ to a schema with all of the $ThreadInvariants$ except $\Delta\ DtosExec$.

$ThreadInvariants$
$\Delta\ DtosExec$
$\Xi\ TaskExist$
$\Xi\ MessageExist$
$\Xi\ MemoryExist$
$\Xi\ PageExist$
$\Xi\ ProcessorExist$
$\Xi\ ProcessorSetExist$
$\Xi\ DeviceExist$
$\Xi\ TaskSuspendCount$
$\Xi\ Kernel$
$\Xi\ RegisteredRights$
$\Xi\ MemoriesAndPorts$
$\Xi\ HostsAndPorts$
$\Xi\ ProcessorsAndPorts$
$\Xi\ SpecialTaskPorts$
$\Xi\ DevicesAndPorts$
$\Xi\ Notifications$
$\Xi\ MessageQueues$
$\Xi\ MemorySystem$
$\Xi\ Messages$
$\Xi\ HostsAndProcessors$
$\Xi\ ProcessorAndProcessorSet$
$\Xi\ TaskAndProcessorSet$
$\Xi\ PortClasses$
$\Xi\ TaskPriority$
$\Xi\ EmulationVector$
$\Xi\ MasterDevicePort$
$\Xi\ HostTime$

9.1.4 General Information

9.1.4.1 Special Ports

The requests **thread_get_special_port** and **thread_set_special_port** each have an input parameter specifying the type of special port to be processed. The following type is used for these input parameters:

$$[THREAD\_SPECIAL\_PORTS]$$

There are two recognized values of this type. They are:

- $Thread\_exception\_port$ — indicates the exception port
- $Thread\_kernel\_port$ — indicates the sself port

We require these two values to be disjoint, but place no restrictions on other values of type $THREAD\_SPECIAL\_PORTS$.

$$\begin{array}{|l}
Thread_exception_port : THREAD_SPECIAL_PORTS \\
Thread_kernel_port : THREAD_SPECIAL_PORTS \\
Recognized_thread_special_ports : \mathbb{P}\ THREAD_SPECIAL_PORTS \\
\hline
\langle Thread_exception_port, Thread_kernel_port \rangle \\
\quad Values_partition\ Recognized_thread_special_ports
\end{array}$$

9.1.4.2 Thread Information Types The request **thread_info** returns an array of information describing a thread. The array used to hold the information is of type $THREAD\_INFO$.

$[THREAD\_INFO]$

There are two recognized types of thread information.

- $Thread\_basic\_info$ — information on execution statistics, execution status and priority
- $Thread\_sched\_info$ — information on scheduling priorities and policies

These two types of information are in the set $THREAD\_INFO\_TYPE$.

$[THREAD\_INFO\_TYPE]$

We require the two values $Thread\_basic\_info$ and $Thread\_sched\_info$ of this type to be disjoint, but place no restrictions on other values of $THREAD\_INFO\_TYPE$.

$$\begin{array}{|l}
Thread_basic_info : THREAD_INFO_TYPE \\
Thread_sched_info : THREAD_INFO_TYPE \\
Recognized_thread_info_types : \mathbb{P}\ THREAD_INFO_TYPE \\
\hline
\langle Thread_basic_info, Thread_sched_info \rangle \\
\quad Values_partition\ Recognized_thread_info_types
\end{array}$$

9.1.4.3 Execution Status Changes Several requests (e.g., **thread_suspend** and **swtch**) can cause the execution of the current thread to be blocked. We describe here the changes that take place when a thread is blocked.

The blocking of a thread results in the thread being swapped out, and another thread moving onto the processor, unless there is nothing else for the processor to swap in. The run states will change for the thread moved off the processor and for the thread moved onto the processor. The thread moved onto the processor is determined by the scheduling algorithm. The algorithm may select the blocking thread in which case the thread remains on the processor. We model this selection of a new thread by the function $Select\_next\_thread$.

Review Note:
This function should be related to the $RunQueue$ which is currently in the specification of the **swtch** request but which should probably be moved to the state chapter. Also, what is the relationship between $Select\_next\_thread$ and $\underline{t}hread\_sched\_priority$? Would it be useful to model this relationship?

$$\begin{array}{|l}
Select_next_thread : PROCESSOR_SET \rightarrowtail THREAD
\end{array}$$

If the scheduling algorithm selects the blocking thread then that thread is marked as not swapped out and not uninterruptibly waiting, and the blocking operation is completed. Otherwise, the new thread selected by the scheduling algorithm receives these markings and, unless the blocking thread is being terminated, the following changes are made to the blocking thread:

1. It is added to $\underline{s}wapped\_threads$.
2. It is marked as not $Running$ if it is not in $\underline{i}dle\_threads$, and it was either marked as
   (a) $Stopped$, but not $Uninterruptible$, or
   (b) $Waiting$.

The schema $ThreadBlock$ describes these changes. The component $blocking\_thread$ is the thread that is being blocked, and $init\_run\_state$ is the run state in effect when the thread blocking occurs. This may differ from the $\underline{r}un\_state$ function depending upon the context in which the blocking occurs. For example, when blocking occurs as part of a **thread_suspend** request the $Stopped$ state will have been added to $\underline{r}un\_state(blocking\_thread)$ to obtain $init\_run\_state$. In the case where the blocking thread is being terminated $blocking\_thread$ is not in the domain of $init\_run\_state$.

Review Note:
The need for $init\_run\_state$ originates in our level of granularity in the specification. There are changes that various requests make to the run state of a thread in preparation for blocking the thread. Since $ThreadBlock$ models only a portion of this processing, we need a way to specify what changes have been made to the run state in the request processing that precedes the blocking.

$ThreadBlock$

$$
\begin{array}{l}
\Delta\ ThreadExecStatus \\
ProcessorAndProcessorSet \\
blocking\_thread : THREAD \\
cpu?? : PROCESSOR \\
init\_run\_state : THREAD \pfun \mathbb{P}\ RUN\_STATES \\
\hline
cpu?? \in \mathrm{dom}\ proc\_assigned\_procset \\
\mathbf{let}\ new\_thread == Select\_next\_thread(proc\_assigned\_procset(cpu??)) \\
\bullet\ new\_thread \notin \underline{s}wapped\_threads' \\
\quad \wedge\ \underline{r}un\_state'(new\_thread) = init\_run\_state(new\_thread) \setminus \{Uninterruptible\} \\
\quad \wedge\ ((new\_thread \neq blocking\_thread \wedge blocking\_thread \notin \mathrm{dom}\ init\_run\_state) \\
\qquad \Rightarrow (blocking\_thread \in \underline{s}wapped\_threads' \\
\qquad\quad \wedge\ \underline{r}un\_state'(blocking\_thread) \\
\qquad\qquad = \mathbf{if}\ blocking\_thread \notin \underline{i}dle\_threads \\
\qquad\qquad\quad \wedge\ (init\_run\_state(blocking\_thread) \\
\qquad\qquad\qquad \cap \{Stopped, Uninterruptible\} = \{Stopped\} \\
\qquad\qquad\qquad \vee\ Waiting \in init\_run\_state(blocking\_thread)) \\
\qquad\qquad \mathbf{then}\ init\_run\_state(blocking\_thread) \setminus \{Running\} \\
\qquad\qquad \mathbf{else}\ init\_run\_state(blocking\_thread))) \\
\quad \wedge\ (\forall\ thread : THREAD \mid thread \notin \{new\_thread, blocking\_thread\} \\
\qquad \bullet\ \underline{r}un\_state'(thread) = init\_run\_state(thread) \\
\qquad \wedge\ thread \in \underline{s}wapped\_threads' \Leftrightarrow thread \in \underline{s}wapped\_threads \\
\qquad \wedge\ thread \in \underline{i}dle\_threads' \Leftrightarrow thread \in \underline{i}dle\_threads)
\end{array}
$$

A request may also wait for a given thread to stop running. The component $stopping\_thread$ is the thread being stopped, and $init\_run\_state$ is defined as for $ThreadBlock$.

$$\begin{array}{l}
\textit{ThreadDoWait} \\
\hline
\Delta\ \textit{ThreadExecStatus} \\
\textit{stopping_thread} : \textit{THREAD} \\
\textit{init_run_state} : \textit{THREAD} \pfun \mathbb{P}\ \textit{RUN_STATES} \\
\hline
\underline{r}\textit{un_state}' = \textit{init_run_state} \\
\qquad \oplus \{\textit{stopping_thread} \mapsto \textit{init_run_state}(\textit{stopping_thread}) \setminus \{\textit{Running}\}\} \\
\underline{s}\textit{wapped_threads}' = \underline{s}\textit{wapped_threads} \\
\underline{i}\textit{dle_threads}' = \underline{i}\textit{dle_threads}
\end{array}$$

Some requests (e.g., **thread_set_state** and **thread_get_state**) must wait for a thread to stop before they can do their work. When they are done modifying or observing the characteristics of the stopped thread they allow the thread to start again. For example, **thread_get_state** stops the thread, examines its machine state (e.g., machine registers) and then allows the thread to run again. The cumulative effect of this sequence of operations might include the side-effect of altering the run state. The run state will contain *Running* when it previously contained neither *Waiting* nor *Stopped*. It will contain *Halted* when it previously contained both *Halted* and *Stopped*. The *Stopped*, *Waiting* and *Uninterruptible* characteristics are unchanged.

Review Note:
We believe that $Halted \Rightarrow Stopped$ at the termination of a request. If this is true then the thread will be halted if and only if it was halted before the request. We also believe that at least one of the states *Running*, *Stopped* and *Waiting* must be contained in the run state. This means that *Running* could be removed from the run state by this operation, but never added.

$$\begin{array}{l}
\textit{ThreadDoWaitThenRelease} \\
\hline
\Delta\ \textit{ThreadExecStatus} \\
\textit{stopping_thread} : \textit{THREAD} \\
\hline
\forall\ \textit{thread} : \textit{THREAD} \mid \textit{thread} \in \operatorname{dom} \underline{r}\textit{un_state} \land \textit{thread} \neq \textit{stopping_thread} \\
\qquad \bullet\ \underline{r}\textit{un_state}'(\textit{thread}) = \underline{r}\textit{un_state}(\textit{thread}) \\
\underline{r}\textit{un_state}'(\textit{stopping_thread}) \cap \{\textit{Stopped}, \textit{Waiting}, \textit{Uninterruptible}\} \\
\qquad = \underline{r}\textit{un_state}(\textit{stopping_thread}) \cap \{\textit{Stopped}, \textit{Waiting}, \textit{Uninterruptible}\} \\
\textit{Halted} \in \underline{r}\textit{un_state}'(\textit{stopping_thread}) \\
\qquad \Leftrightarrow \{\textit{Halted}, \textit{Stopped}\} \subseteq \underline{r}\textit{un_state}(\textit{stopping_thread}) \\
\textit{Running} \in \underline{r}\textit{un_state}'(\textit{stopping_thread}) \\
\qquad \Leftrightarrow \underline{r}\textit{un_state}(\textit{stopping_thread}) \cap \{\textit{Waiting}, \textit{Stopped}\} = \varnothing
\end{array}$$

9.1.4.4 Parameter Packaging Functions

When invoking a kernel request, the following functions package the input parameters into a message body:

$Name\_and\_number\_to\_text : NAME \times \mathbb{Z} \longrightarrow MESSAGE\_BODY$
$Name\_to\_text : NAME \longrightarrow MESSAGE\_BODY$
$Number\_and\_boolean\_to\_text : \mathbb{Z} \times BOOLEAN \longrightarrow MESSAGE\_BODY$
$Policy\_and\_data\_to\_text : SCHED\_POLICY \times SCHED\_POLICY\_DATA$
$\quad \longrightarrow MESSAGE\_BODY$
$Sample\_type\_set\_to\_text : \mathbb{P}\, SAMPLE\_TYPES \longrightarrow MESSAGE\_BODY$
$Sequence\_number\_to\_text : \mathbb{N} \longrightarrow MESSAGE\_BODY$
$Thread\_info\_type\_and\_count\_to\_text : THREAD\_INFO\_TYPE \times \mathbb{N}$
$\quad \longrightarrow MESSAGE\_BODY$
$Thread\_set\_state\_params\_to\_text :$
$\quad THREAD\_STATE\_INFO\_TYPES \times THREAD\_STATE\_INFO \times \mathbb{N}$
$\qquad \longrightarrow MESSAGE\_BODY$
$Thread\_special\_port\_and\_name\_to\_text : THREAD\_SPECIAL\_PORTS \times NAME$
$\quad \longrightarrow MESSAGE\_BODY$
$Thread\_special\_ports\_to\_text : THREAD\_SPECIAL\_PORTS \longrightarrow MESSAGE\_BODY$
$Thread\_state\_info\_type\_and\_number\_to\_text : THREAD\_STATE\_INFO\_TYPES \times \mathbb{N}$
$\quad \longrightarrow MESSAGE\_BODY$

When creating a reply message from a request, the following functions package the output parameters into a kernel reply:

$Return\_capability : Capability \longrightarrow KERNEL\_REPLY$
$Return\_sample\_cnt : \mathbb{N} \longrightarrow KERNEL\_REPLY$
$Return\_samples : (\mathbb{N} \times (\mathrm{seq}\, SAMPLE) \times \mathbb{Z}) \longrightarrow KERNEL\_REPLY$
$Return\_thread\_info : THREAD\_INFO \times \mathbb{N} \longrightarrow KERNEL\_REPLY$
$Return\_thread\_state\_info : THREAD\_STATE\_INFO \times \mathbb{N} \longrightarrow KERNEL\_REPLY$
$Return\_ticks : \mathbb{N}_1 \longrightarrow KERNEL\_REPLY$

When receiving a reply message from the kernel the following functions unpack the message body to obtain the output parameters (including the return status):

$Text\_to\_count\_and\_status : MESSAGE\_BODY \longrightarrow (\mathbb{N} \times KERNEL\_RETURN)$
$Text\_to\_info\_and\_count\_and\_status : MESSAGE\_BODY$
$\quad \longrightarrow (THREAD\_INFO \times \mathbb{N} \times KERNEL\_RETURN)$
$Text\_to\_name\_and\_status : MESSAGE\_BODY \longrightarrow (NAME \times KERNEL\_RETURN)$
$Text\_to\_seqno\_and\_PCs\_and\_count\_and\_status : MESSAGE\_BODY$
$\quad \longrightarrow (\mathbb{N} \times \mathrm{seq}\, SAMPLE \times \mathbb{Z} \times KERNEL\_RETURN)$
$Text\_to\_state\_and\_count\_and\_status : MESSAGE\_BODY$
$\quad \longrightarrow (THREAD\_STATE\_INFO \times \mathbb{N} \times KERNEL\_RETURN)$
$Text\_to\_status : MESSAGE\_BODY \longrightarrow KERNEL\_RETURN$
$Text\_to\_ticks\_and\_status : MESSAGE\_BODY \longrightarrow (\mathbb{N}_1 \times KERNEL\_RETURN)$

9.1.4.5 Destroying a Port

Review Note:
The following may fit more naturally in the port requests chapter.

The following schema describes the destruction of a port. This is required to describe the **thread_terminate** request. The port destroyed is removed from the set of existing ports, and all send, receive, and send-once rights to this port are removed from all port name spaces. New dead names are created for all previous send or send-once rights to this port. Note that the creation of notifications when these names turn into dead names should be added to this schema.

Editorial Note:
Destruction of a port representing a message queue for IPC can cause a chain reaction not represented in this schema. Whether the same chain reaction is possible for ports representing kernel objects (as the schema is used in this chapter) is unclear.

$$
\begin{array}{l}
\textit{PortDestroy} \\
\hline
\Delta\, Ipc \\
port : PORT \\
\hline
\underline{p}ort\_exists' = \underline{p}ort\_exists \setminus \{port\} \\
\underline{p}ort\_right\_rel' = \underline{p}ort\_right\_rel \\
\quad \setminus \{task : TASK;\ name : NAME;\ right : RIGHT;\ i : \mathbb{N}_1 \\
\qquad \bullet (task, port, name, right, i)\} \\
\underline{m}ake\_send\_count' = \{port\} \lhd \underline{m}ake\_send\_count \\
\underline{d}ead\_right\_rel' = \underline{d}ead\_right\_rel \\
\quad \cup \{task : TASK;\ name : NAME \\
\qquad \mid named\_port(task, name) = port \\
\qquad \bullet (task, name, 1)\}
\end{array}
$$

9.1.4.6 Miscellaneous The function $Thread\_port\_to\_s\_right$ takes a port and returns a send right to the port.

$$
\begin{array}{l}
Thread\_port\_to\_s\_right : PORT \rightarrowtail\!\!\!\!\!+ Capability \\
\hline
\forall\, port : PORT \\
\bullet\ (Thread\_port\_to\_s\_right(port)).right = Send \\
\wedge\ (Thread\_port\_to\_s\_right(port)).port = port
\end{array}
$$

The function $Thread\_state\_count$ returns the size of a given type of thread state information.

$$
Thread\_state\_count : THREAD\_STATE\_INFO\_TYPES \longrightarrow \mathbb{N}
$$

9.1.5 Kernel Processing

The kernel performs processing for a thread request only when it detects a break indicating that a request has been received through a port of the appropriate class, Pc_task or Pc_thread. If the specified service port no longer exists, then a $Kern\_invalid\_argument$ status code is returned.

$$
\begin{array}{|l}
\hline
\textit{ProcessThreadRequestBadAux} \\
\textit{ProcessRequest} \\
\Xi\ \textit{Mach} \\
\textit{reply\_to\_port}! : \mathbb{P}\ \textit{PORT} \\
\textit{reply}! : \textit{KERNEL\_REPLY} \\
\textit{return}! : \textit{KERNEL\_RETURN} \\
\hline
((\textit{pc}? = \textit{Pc\_thread} \land \textit{operation}? \in \textit{Thread\_thread\_port\_ops} \\
\quad \land\ \textit{service\_port}? \notin \operatorname{dom} \textit{self\_thread}) \\
\lor\ (\textit{pc}? = \textit{Pc\_task} \land \textit{operation}? \in \textit{Thread\_task\_port\_ops} \\
\quad \land\ \textit{service\_port}? \notin \operatorname{dom} \textit{self\_task})) \\
\\
\textit{reply\_to\_port}! = \textit{reply\_to\_port}? \\
\textit{return}! = \textit{Kern\_invalid\_argument} \\
\hline
\end{array}
$$

$$\textit{ProcessThreadRequestBad} \mathrel{\widehat=} \textit{ProcessThreadRequestBadAux} \gg \textit{RequestNoOp}$$

Otherwise, the kernel processes the request. In this case, we use the following schema to represent the parameters to thread requests which are processed via thread ports:

Editorial Note:
$\textit{flavor}?$ is omitted because it is used with two different types in different requests.

$$
\begin{array}{|l}
\hline
\textit{ThreadParameters} \\
\textit{data}? : \textit{SCHED\_POLICY\_DATA} \\
\textit{new\_state}? : \textit{THREAD\_STATE\_INFO} \\
\textit{new\_state\_cnt}? : \mathbb{N} \\
\textit{old\_state\_cnt}? : \mathbb{N} \\
\textit{policy}? : \textit{SCHED\_POLICY} \\
\textit{priority}? : \mathbb{Z} \\
\textit{procset}? : \textit{PROCESSOR\_SET} \\
\textit{seqno}? : \mathbb{N} \\
\textit{set\_max}? : \textit{BOOLEAN} \\
\textit{special\_port}? : \textit{PORT} \\
\textit{task}? : \textit{TASK} \\
\textit{thread}? : \textit{THREAD} \\
\textit{target\_thread}? : \textit{THREAD} \\
\textit{thread\_infoCnt}? : \mathbb{N} \\
\textit{which\_port}? : \textit{THREAD\_SPECIAL\_PORTS} \\
\hline
\end{array}
$$

The interpretations of the components of these schemas are:

- $\textit{data}?$ — policy specific data used with the scheduling policy to determine the scheduling priority of a thread (**thread_policy**)
- $\textit{flavor}?$ — specific type of information (**thread_info**) or state information (**thread_get_state**, **thread_set_state** and **thread_set_state_secure**)
- $\textit{new\_state}?$ — state information to be assigned to a thread (**thread_set_state** and **thread_set_state_secure**)

- $new\_state\_cnt?$ — amount of storage provided by a client to hold state information to be assigned to a thread (**thread_set_state** and **thread_set_state_secure**)
- $old\_state\_cnt?$ — amount of storage provided by a client to hold state information (**thread_get_state**)
- $policy?$ — desired scheduling policy (**thread_policy**)
- $priority?$ — desired priority for a thread (**thread_priority** and **thread_max_priority**)
- $procset?$ — desired processor set for a thread (**thread_assign**); the control port for the processor set to which a thread is currently assigned (**thread_max_priority**)
- $seqno?$ — the sequence number of the first sample that should be returned (**thread_get_sampled_pcs**)
- $set\_max?$ — a flag indicating whether the maximum priority should be reset when the $priority$ is changed (**thread_priority**)
- $special\_port?$ — a port specified by the client to become the special port for the target thread (for **thread_set_special_port**)
- $task?$ — the target task for the request (**thread_create** and **thread_create_secure**)
- $target\_thread?$ — the target thread for the request (an alternative name for $thread?$ that is used in some requests)
- $thread?$ — the target thread for the request
- $thread\_infoCnt?$ — amount of storage provided by a client to hold information on a thread (**thread_info**)
- $which\_port?$ — the type of special port specified by the client (**thread_get_special_port** and **thread_set_special_port**)

The following schema maps a message sent to a thread port to a value of type $ThreadParameters$:

$ThreadMessageToThreadParameters$

$Request?$
$SpecialThreadPorts$
$ThreadParameters$

$pc? = Pc\_thread$
$operation? \in Thread\_thread\_port\_ops$
$service\_port? \in \mathrm{dom}\, self\_thread$
$thread? = self\_thread(service\_port?)$
$target\_thread? = thread?$

Review Note:
$ThreadInvariants$ really belongs in the state changes schemas rather than here in the message processing. What I want here is Ξalmosteverything. The same goes for the use a couple schemas further down.

ProcessThreadViaThreadPortRequestGood
ProcessRequest
ThreadInvariants
ThreadMessageToThreadParameters

Similarly, we use the following function to map a message sent to a task port to a value of type *ThreadParameters*:

TaskMessageToThreadParameters
Request?
SpecialTaskPorts
ThreadParameters

$pc? = Pc\_task$
$operation? \in Thread\_task\_port\_ops$
$service\_port? \in \mathrm{dom}\, self\_task$
$task? = self\_task(service\_port?)$

ProcessThreadViaTaskPortRequestGood
ProcessRequest
ThreadInvariants
TaskMessageToThreadParameters

We now describe the individual thread requests.

9.2 thread_abort

The request **thread_abort** helps to cleanly stop a thread by interrupting page faults and any **mach_msg** calls in progress by the thread. It causes an interrupt return code to be returned from any system trap in progress on behalf of the thread (even though the execution of the trap may finish). It also aborts any priority depressions. Note that **thread_abort** does not suspend a thread. If the thread did not already have the *Stopped* state, then at the conclusion of a **thread_abort** request it is neither *Stopped* nor *Halted*.

9.2.1 Client Interface

kern_return_t **thread_abort**
(mach_port_t *target_thread_name*);

9.2.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_abort** request:

- *target_thread_name?* — the client's name for the thread to which the abort will be applied

ThreadAbortClientInputs
$target\_thread\_name? : NAME$

A **thread_abort** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_abort\_id$ and has no body.

$$
\begin{array}{l}
\textit{InvokeThreadAbort} \\
\hline
\textit{InvokeMachMsg} \\
\textit{ThreadAbortClientInputs} \\
\hline
name? = target\_thread\_name? \\
operation? = Thread\_abort\_id
\end{array}
$$

9.2.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_abort** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadAbortClientOutputs} \\
\hline
return! : KERNEL\_RETURN
\end{array}
$$

$$
\begin{array}{l}
\textit{ThreadAbortReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{ThreadAbortClientOutputs} \\
\hline
return! = Text\_to\_status(msg\_body)
\end{array}
$$

9.2.2 Kernel Interface

9.2.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_abort** request:

- $target\_thread?$ — the thread to which the abort will be applied

$$
\begin{array}{l}
\textit{ThreadAbortInputs} \\
\hline
target\_thread? : THREAD
\end{array}
$$

9.2.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_abort** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadAbortOutputs} \\
\hline
return! : KERNEL\_RETURN
\end{array}
$$

9.2.3 Request Criteria

The following criteria are defined for the **thread_abort** request.

- **C1 —** The parameter $target\_thread?$ is the client thread (i.e., the thread currently active on the CPU).

$$
\begin{array}{|l}
\multicolumn{1}{l}{C1ThreadAbortClientThread} \\
ThreadsAndProcessors \\
cpu?? : PROCESSOR \\
target\_thread? : THREAD \\
\hline
cpu?? \in \mathrm{dom}\ \underline{a}ctive\_thread \\
\\
target\_thread? = \underline{a}ctive\_thread(cpu??) \\
\hline
\end{array}
$$

$$
NotC1ThreadAbortClientThread \\
\quad \widehat{=}\ ThreadsAndProcessors \land \lnot\ C1ThreadAbortClientThread
$$

9.2.4 Return Values

Table 17 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$	C1
$Kern\_invalid\_argument$	T
$Kern\_success$	F

Table 17: Return Values for **thread_abort**

Review Note:
thread_abort can return $Kern\_aborted$ when

- there is a cycle of halt operations, or
- the client thread is interrupted while waiting for the target thread to halt.

IPC will convert $Kern\_aborted$ to an IPC interrupted error code. This behavior is not modeled.

$$
\begin{array}{|l}
\multicolumn{1}{l}{RVThreadAbortInvalidArgument} \\
C1ThreadAbortClientThread \\
ThreadAbortOutputs \\
\hline
return! = Kern\_invalid\_argument \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{RVThreadAbortGood} \\
NotC1ThreadAbortClientThread \\
ThreadAbortOutputs \\
\hline
return! = Kern\_success \\
\hline
\end{array}
$$

9.2.5 State Changes

A successful **thread_abort** request will interrupt page faults and message primitive calls in use by the thread. The thread will resume execution at the point of return from the interrupted system call. This will occur upon return from this request unless the thread is in a *Stopped* state, in which case it will occur when the thread is resumed via **thread_resume**.

Review Note:
The granularity of the FTLS is not fine enough to model the interruption of page faults and message primitive calls in use by the thread.

As stated above, **thread_abort** does not suspend a thread. If the thread does not already have the *Stopped* state, then at the conclusion of a **thread_abort** request it is neither *Stopped* nor *Halted*. If the thread has been previously *Stopped*, the thread will remain *Stopped* upon completion of the **thread_abort** request until it is resumed. The thread will also be *Halted* at this point since **thread_abort** has insured that it is stopped at a clean point. A thread that is not already *Stopped* and is not *Waiting* will have *Running* added to its run state by **thread_abort** (assuming it is not already there).

$$
\begin{array}{l}
\textit{ThreadAbortExecStatus} \\
\hline
\Delta\ \textit{ThreadExecStatus} \\
\textit{target\_thread}? : \textit{THREAD} \\
\hline
\forall\, \textit{thread} : \textit{THREAD} \mid \textit{thread} \in \operatorname{dom} \underline{r}\textit{un\_state} \wedge \textit{thread} \neq \textit{target\_thread}? \\
\qquad \bullet\ \underline{r}\textit{un\_state}'(\textit{thread}) = \underline{r}\textit{un\_state}(\textit{thread}) \\
\underline{r}\textit{un\_state}'(\textit{target\_thread}?) \cap \{\textit{Stopped}, \textit{Waiting}, \textit{Uninterruptible}\} \\
\qquad = \underline{r}\textit{un\_state}(\textit{target\_thread}?) \cap \{\textit{Stopped}, \textit{Waiting}, \textit{Uninterruptible}\} \\
\textit{Halted} \in \underline{r}\textit{un\_state}'(\textit{target\_thread}?) \Leftrightarrow \textit{Stopped} \in \underline{r}\textit{un\_state}(\textit{target\_thread}?) \\
\textit{Running} \in \underline{r}\textit{un\_state}'(\textit{target\_thread}?) \\
\qquad \Leftrightarrow \underline{r}\textit{un\_state}(\textit{target\_thread}?) \cap \{\textit{Waiting}, \textit{Stopped}\} = \varnothing \\
\underline{s}\textit{wapped\_threads}' = \underline{s}\textit{wapped\_threads} \\
\underline{i}\textit{dle\_threads}' = \underline{i}\textit{dle\_threads} \\
\underline{t}\textit{hread\_suspend\_count}' = \underline{t}\textit{hread\_suspend\_count} \\
\underline{t}\textit{hreads\_wired}' = \underline{t}\textit{hreads\_wired} \\
\hline
\end{array}
$$

The sending of the return from this request means that the thread has received an interrupt return code from the program it was executing. It follows from the specification for the processing of invokable requests and **mach_msg** that the receipt of the return of *Kern_success* from this request will therefore occur only if the thread is not in *Stopped* state. If the target thread is in a stopped state, it will receive the return value when (and if) it is resumed (via **thread_resume**).

Any priority depression is also aborted. This returns the priority of the thread to its value before the depression. Note that the scheduling priority may also change, but since we do not have enough detail in our model to compute its value we will leave it unspecified.

$$
\begin{array}{l}
\textbf{ThreadAbortPriority} \\
\hline
\Delta\ ThreadPri \\
target\_thread? : THREAD \\
\hline
target\_thread? \in \operatorname{dom} \underline{p}riority\_before\_depression \\
\\
\underline{t}hread\_priority' = \underline{t}hread\_priority \\
\qquad \oplus \{target\_thread? \mapsto \underline{p}riority\_before\_depression(target\_thread?)\} \\
\underline{t}hread\_max\_priority' = \underline{t}hread\_max\_priority \\
\underline{d}epressed\_threads' = \underline{d}epressed\_threads \setminus \{target\_thread?\} \\
\underline{p}riority\_before\_depression' = \underline{p}riority\_before\_depression
\end{array}
$$

When the target thread resumes execution it will be at the return point from any interrupt of trap it might have been executing. The component at_call_return represents the address at which execution will resume if the thread is resumed.

Editorial Note:
Our model is not detailed enough to formally describe the value of at_call_return.

$$
\begin{array}{l}
\textbf{ThreadAbortState} \\
\hline
ThreadInvariants \\
ThreadAbortExecStatus \\
ThreadAbortPriority \\
\Xi\ ThreadExist \\
\Xi\ PortExist \\
\Delta\ Threads \\
\Xi\ TasksAndThreads \\
\Xi\ ThreadSchedPolicy \\
\Delta\ ThreadInstruction \\
\Delta\ Events \\
\Xi\ PortNameSpace \\
\Xi\ SpecialPurposePorts \\
target\_thread? : THREAD \\
at\_call\_return : VIRTUAL\_ADDRESS \\
\hline
\underline{i}nstruction\_pointer' = \underline{i}nstruction\_pointer \oplus \{target\_thread? \mapsto at\_call\_return\}
\end{array}
$$

Review Note:
How can we represent here the fact that the execution of messages and traps might not be completed? There might be some delay in halting the thread. Is this important?

9.2.6 Complete Request

The following schema defines the general form of a **thread_abort** request.

$$
\begin{array}{l}
\textbf{ProcessingThreadAbort} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_abort\_id
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$ThreadAbortGood \mathrel{\widehat=} (RVThreadAbortGood \land ThreadAbortState) \\ \quad \gg RequestReturnOnlyStatus$$

An unsuccessful request returns an error status.

$$ThreadAbortBad \mathrel{\widehat=} RVThreadAbortInvalidArgument \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadAbort \mathrel{\widehat=} (ThreadAbortGood \lor ThreadAbortBad) \setminus (at\_call\_return)$$

The full specification for kernel processing of a validated **thread_abort** request consists of processing the request followed by its execution.

$$ThreadAbort \mathrel{\widehat=} ProcessingThreadAbort \mathbin{⨾} ExecuteThreadAbort$$

9.3 **thread_create** and **thread_create_secure**

The requests **thread_create** and **thread_create_secure** create a new thread within an existing task. The name of a send right to the kernel port of the new thread is returned. The **thread_create_secure** request (which is used in the secure initiation of threads within a task) expects the parent task to have task creation state Tcs_task_empty (see Section 5.7). It modifies the state to $Tcs\_thread\_created$.

9.3.1 Client Interface

```
kern_return_t thread_create
              (mach_port_t          parent_task_name,
               mach_port_t*         child_thread_name);

kern_return_t thread_create_secure
              (mach_port_t          parent_task_name,
               mach_port_t*         child_thread_name);
```

9.3.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_create** or **thread_create_secure** request:

- $parent\_task\_name?$ — the client's name for the task that will be the parent for the newly created thread

$$\begin{array}{|l} \hline ThreadCreateClientInputs \\ parent\_task\_name? : NAME \\ \hline \end{array}$$

A **thread_create** request is invoked by sending a message to the port indicated by $parent\_task\_name?$ that has the operation field set to $Thread\_create\_id$ and has no body.

$$
\begin{array}{l}
\textit{InvokeThreadCreate} \\
\hline
\textit{InvokeMachMsg} \\
\textit{ThreadCreateClientInputs} \\
\hline
name? = parent\_task\_name? \\
operation? = Thread\_create\_id
\end{array}
$$

A **thread_create_secure** request is invoked by sending a message to the port indicated by $parent\_task\_name?$ that has the operation field set to $Thread\_create\_secure\_id$ and has no body.

$$
\begin{array}{l}
\textit{InvokeThreadCreateSecure} \\
\hline
\textit{InvokeMachMsg} \\
\textit{ThreadCreateClientInputs} \\
\hline
name? = parent\_task\_name? \\
operation? = Thread\_create\_secure\_id
\end{array}
$$

9.3.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_create** or **thread_create_secure** request:

- $child\_thread\_name!$ — the name of a send right to the kernel port of the new thread
- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadCreateClientOutputs} \\
\hline
child\_thread\_name! : NAME \\
return! : KERNEL\_RETURN
\end{array}
$$

$$
\begin{array}{l}
\textit{ThreadCreateReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{ThreadCreateClientOutputs} \\
\hline
(child\_thread\_name!, return!) = Text\_to\_name\_and\_status(msg\_body)
\end{array}
$$

9.3.2 Kernel Interface

9.3.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_create** or **thread_create_secure** request:

- $\underline{p}arent\_task?$ — the task that will be the parent for the newly created thread

$$
\begin{array}{l}
\textit{ThreadCreateInputs} \\
\hline
\underline{p}arent\_task? : TASK
\end{array}
$$

9.3.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_create** or **thread_create_secure** request:

- $child\_thread!$ — the new thread
- $return!$ — the status of the request

$$\begin{array}{|l}
\textit{ThreadCreateOutputs} \\
\hline
child_thread! : THREAD \\
return! : KERNEL_RETURN \\
\hline
\end{array}$$

Upon completion of the processing of a **thread_create** or **thread_create_secure** request a reply message is built from the output parameters. The reply message will contain a send right for the created thread's kernel port.

$$\begin{array}{|l}
\textit{ThreadCreateReply} \\
\hline
RequestReturn \\
child_thread? : THREAD \\
\hline
reply? = Return_capability(Thread_port_to_s_right(thread_sself(child_thread?))) \\
\hline
\end{array}$$

9.3.3 Request Criteria

The following criteria are defined for the **thread_create** and **thread_create_secure** requests.

- **C1 —** The kernel has the necessary resources available to create the thread. We do not actually model the consumption of resources by the kernel. So, we will use the set $Resources\_available\_to\_create\_thread$ to indicate the set of states where there are sufficient resources to create a thread.

$$Resources\_available\_to\_create\_thread : \mathbb{P}\ DtosExec$$

$$\begin{array}{|l}
\textit{C1ThreadCreateResourcesAvailable} \\
\hline
DtosExec \\
\hline
\theta DtosExec \in Resources_available_to_create_thread \\
\hline
\end{array}$$

$$NotC1ThreadCreateResourcesAvailable \mathrel{\widehat=} DtosExec \land \neg\ C1ThreadCreateResourcesAvailable$$

- **C2 —** The task creation state of the parent task must be Tcs_task_empty. This criterion applies only to the **thread_create_secure** request.

$$\begin{array}{|l}
\textit{C2ThreadCreateSecureTaskEmpty} \\
\hline
TaskCreationState \\
\underline{p}arent_task? : TASK \\
\hline
\underline{p}arent_task? \in \mathrm{dom}\ \underline{t}ask_creation_state \\
\\
\underline{t}ask_creation_state(\underline{p}arent_task?) = Tcs_task_empty \\
\hline
\end{array}$$

$$NotC2ThreadCreateSecureTaskEmpty \mathrel{\widehat=} TaskCreationState \land \neg\ C2ThreadCreateSecureTaskEmpty$$

9.3.4 Return Values

Table 18 describes the values returned at the completion of the **thread_create** request and the conditions under which each value is returned. The value $thread$ is the newly created thread (see the State Changes section). The design does not specify the value of $child\_thread!$ when an error occurs. It depends on the implementation, and we leave it unspecified.

$child\_thread!$	$return!$	C1
$thread$	$Kern\_success$	T
—	$Kern\_resource\_shortage$	F

Table 18: Return Values for **thread_create**

$RVThreadCreateGood$
$C1ThreadCreateResourcesAvailable$
$ThreadCreateOutputs$
$thread : THREAD$

$child\_thread! = thread$
$return! = Kern\_success$

$RVThreadCreateResourceShortage$
$NotC1ThreadCreateResourcesAvailable$
$ThreadCreateOutputs$

$return! = Kern\_resource\_shortage$

Table 19 describes the values returned at the completion of the **thread_create_secure** request and the conditions under which each value is returned. In the case where both C1 and C2 are false we assume that $Kern\_insufficient\_permission$ is returned.

Review Note:
In the prototype the conditions are checked in the order C2, C1.

$child\_thread!$	$return!$	C1	C2
$thread$	$Kern\_success$	T	T
—	$Kern\_resource\_shortage$	F	T
—	$Kern\_insufficient\_permission$	-	F

Table 19: Return Values for **thread_create_secure**

$RVThreadCreateSecureGood$
$C1ThreadCreateResourcesAvailable$
$C2ThreadCreateSecureTaskEmpty$
$ThreadCreateOutputs$
$thread : THREAD$

$child\_thread! = thread$
$return! = Kern\_success$

$$
\begin{array}{l}
\textit{RVThreadCreateSecureResourceShortage} \\
\hline
NotC1ThreadCreateResourcesAvailable \\
C2ThreadCreateSecureTaskEmpty \\
ThreadCreateOutputs \\
\hline
return! = Kern\_resource\_shortage
\end{array}
$$

$$
\begin{array}{l}
\textit{RVThreadCreateSecureInsufficientPermission} \\
\hline
NotC2ThreadCreateSecureTaskEmpty \\
ThreadCreateOutputs \\
\hline
return! = Kern\_insufficient\_permission
\end{array}
$$

9.3.5 State Changes

A successful **thread_create** or **thread_create_secure** request creates a new thread. The OSF documentation for this request states that, in addition, a send right to the thread's kernel port is given to the containing task. This is not shown explicitly here. We believe that the existence of a new thread self port is an "implicit" send right, not in the port name space (and not usable) for the containing task until the thread executes a **mach_thread_self** request.

The creation of a new thread affects much of the state information associated with threads. We will consider each type of state information individually. We first define the things that do not change in a successful **thread_create** or **thread_create_secure** request. Note that the port name space of the receiving task on the reply port for this request will change after the invokable request created by the schema *Return* is processed, and not immediately upon completion of this request.

$$
\begin{array}{l}
\textit{ThreadCreateInvariants} \\
\hline
ThreadInvariants \\
\Xi\ PortNameSpace
\end{array}
$$

A new thread is created and added to the list of threads associated with the parent task.

$$
\begin{array}{l}
\textit{ThreadCreateTasksAndThreads} \\
\hline
\Delta\ ThreadExist \\
\Delta\ TasksAndThreads \\
parent\_task? : TASK \\
thread : THREAD \\
\hline
thread \notin thread\_exists \\
\\
thread\_exists' = thread\_exists \cup \{thread\} \\
task\_thread\_rel' = task\_thread\_rel \cup \{(parent\_task?, thread)\}
\end{array}
$$

A newly created thread takes its maximum priority to be the lower of the following two priorities:

- the maximum priority of the processor set to which it is assigned, or

- the $Base\_user\_priority$ constant.

It takes its priority to be the lower of its parent task's priority and its own maximum priority. No other thread's priorities change due solely to the creation of this thread.

$$
\begin{array}{l}
\textit{ThreadCreatePriority} \\
\hline
\Delta\ ThreadPri \\
TaskPriority \\
\Delta\ ThreadAndProcessorSet \\
\underline{p}arent\_task? : TASK \\
thread : THREAD \\
\hline
thread \in \mathrm{dom}\, thread\_assigned\_to' \\
thread\_assigned\_to'(thread) \in \mathrm{dom}\, \underline{p}s\_max\_priority \\
\underline{p}arent\_task? \in \mathrm{dom}\, \underline{t}ask\_priority \\
\\
\underline{t}hread\_max\_priority' = \underline{t}hread\_max\_priority \\
\quad \cup \{thread \mapsto Lowest\_priority(\{\underline{p}s\_max\_priority(thread\_assigned\_to'(thread)), \\
\qquad Base\_user\_priority\})\} \\
\underline{t}hread\_priority' = \underline{t}hread\_priority \\
\quad \cup \{thread \mapsto Lowest\_priority(\{\underline{t}ask\_priority(\underline{p}arent\_task?), \\
\qquad \underline{t}hread\_max\_priority'(thread)\})\} \\
\mathrm{dom}\, \underline{t}hread\_sched\_priority' = \mathrm{dom}\, \underline{t}hread\_sched\_priority \cup \{thread\} \\
\underline{t}hread\_sched\_priority \subset \underline{t}hread\_sched\_priority' \\
\underline{d}epressed\_threads' = \underline{d}epressed\_threads \\
\underline{p}riority\_before\_depression' \\
\quad = \underline{p}riority\_before\_depression \cup \{thread \mapsto \underline{t}hread\_priority'(thread)\} \\
\hline
\end{array}
$$

The new thread's scheduling policy is $Timeshare$. Since the $Timeshare$ policy does not require any scheduling policy data, there is no change to$\underline{t}hread_sched_policy_data$.

$$
\begin{array}{l}
\textit{ThreadCreateSchedPolicy} \\
\hline
\Delta\ ThreadSchedPolicy \\
thread : THREAD \\
\hline
\underline{t}hread\_sched\_policy' = \underline{t}hread\_sched\_policy \cup \{thread \mapsto Timeshare\} \\
\underline{t}hread\_sched\_policy\_data' = \underline{t}hread\_sched\_policy\_data \\
\hline
\end{array}
$$

The thread is created in a $Stopped$ run state, and it is swapped out. Its suspend count is one larger than the suspend count of its parent task.

$$
\begin{array}{l}
\textbf{ThreadCreateExecStatus} \\
\hline
\Delta\ ThreadExecStatus \\
TaskSuspendCount \\
thread : THREAD \\
parent\_task? : TASK \\
\hline
parent\_task? \in \operatorname{dom} task\_suspend\_count \\
\\
run\_state' = run\_state \cup \{thread \mapsto \{Stopped\}\} \\
swapped\_threads' = swapped\_threads \cup \{thread\} \\
thread\_suspend\_count' = thread\_suspend\_count \\
\qquad \cup \{thread \mapsto task\_suspend\_count(parent\_task?) + 1\} \\
threads\_wired' = threads\_wired \\
\hline
\end{array}
$$

All of the thread's timing statistics are set to zero.

$$
\begin{array}{l}
\textbf{ThreadCreateStatistics} \\
\hline
\Delta\ ThreadStatistics \\
thread : THREAD \\
\hline
user\_time' = user\_time \cup \{thread \mapsto 0\} \\
system\_time' = system\_time \cup \{thread \mapsto 0\} \\
cpu\_time' = cpu\_time \cup \{thread \mapsto 0\} \\
sleep\_time' = sleep\_time \cup \{thread \mapsto 0\} \\
\hline
\end{array}
$$

A new self port is created for the thread. This port is assigned to be the kernel (sself) port as well. There is no exception port assigned to the thread.

$$
\begin{array}{l}
\textbf{ThreadCreateSpecialPorts} \\
\hline
\Delta\ PortExist \\
\Delta\ SpecialThreadPorts \\
thread : THREAD \\
port : PORT \\
\hline
port \notin port\_exists \\
\\
port\_exists' = port\_exists \cup \{port\} \\
thread\_self' = thread\_self \cup \{thread \mapsto port\} \\
thread\_sself' = thread\_sself \cup \{thread \mapsto port\} \\
thread\_eport' = thread\_eport \\
\hline
\end{array}
$$

The thread will be assigned to the processor set to which its parent task is assigned.

$$
\begin{array}{l}
\textbf{\textit{ThreadCreateThreadAndProcessorSet}} \\
\hline
\Delta\ ThreadAndProcessorSet \\
TaskAndProcessorSet \\
\underline{p}arent\_task? : TASK \\
\underline{t}hread : THREAD \\
\hline
\underline{p}arent\_task? \in \operatorname{dom} task\_assigned\_to \\
\\
\underline{t}hread\_assignment\_rel' = \underline{t}hread\_assignment\_rel \\
\qquad \cup \{(thread, task\_assigned\_to(\underline{p}arent\_task?))\} \\
\underline{e}nabled\_sp' = \underline{e}nabled\_sp \\
\underline{p}s\_max\_priority' = \underline{p}s\_max\_priority \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{ThreadCreateState}} \\
\hline
\Delta\ Threads \\
ThreadCreateInvariants \\
ThreadCreateTasksAndThreads \\
ThreadCreatePriority \\
ThreadCreateSchedPolicy \\
ThreadCreateExecStatus \\
ThreadCreateStatistics \\
ThreadCreateSpecialPorts \\
ThreadCreateThreadAndProcessorSet \\
\Delta\ SpecialPurposePorts \\
\hline
\end{array}
$$

For the **thread_create_secure** request the task creation state of the parent task is changed to $Tcs\_thread\_created$. There is no change to the task creation state of the parent task for a **thread_create** request.

$$
\begin{array}{l}
\textbf{\textit{ThreadCreateSecureState}} \\
\hline
\Delta\ TaskCreationState \\
\underline{p}arent\_task? : TASK \\
operation? : OPERATION \\
\hline
(operation? = Thread\_create\_secure\_id \\
\qquad \wedge\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state \\
\qquad\qquad \oplus \{\underline{p}arent\_task? \mapsto Tcs\_thread\_created\}) \\
\vee\ (operation? = Thread\_create\_id \\
\qquad \wedge\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state) \\
\hline
\end{array}
$$

The new port gets a SID based upon the parent task.

$$
\begin{array}{l}
\textbf{\textit{ThreadCreateDtosState}} \\
\hline
\Delta\ PortSid \\
SubjectSid \\
port : PORT \\
\underline{p}arent\_task? : TASK \\
\hline
\underline{p}ort\_sid' = \underline{p}ort\_sid \cup \{port \mapsto Thread\_port\_sid(\underline{t}ask\_sid(\underline{p}arent\_task?))\} \\
\hline
\end{array}
$$

9.3.6 Complete Request

The following schemas define the general form of the **thread_create** and **thread_create_secure** requests.

$$\begin{array}{|l}\hline \textit{ProcessingThreadCreate} \\ \textit{ProcessThreadViaTaskPortRequestGood} \\ \hline \textit{operation}? = \textit{Thread\_create\_id} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{ProcessingThreadCreateSecure} \\ \textit{ProcessThreadViaTaskPortRequestGood} \\ \hline \textit{operation}? = \textit{Thread\_create\_secure\_id} \\ \hline\end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$\begin{aligned}
&\textit{ThreadCreateGood} \mathrel{\widehat=} \\
&\quad (\textit{RVThreadCreateGood} \land \textit{ThreadCreateState} \\
&\qquad \land \textit{ThreadCreateSecureState} \land \textit{ThreadCreateDtosState}) \\
&\quad \gg \textit{ThreadCreateReply} \\
&\textit{ThreadCreateSecureGood} \mathrel{\widehat=} \\
&\quad (\textit{RVThreadCreateSecureGood} \land \textit{ThreadCreateState} \\
&\qquad \land \textit{ThreadCreateSecureState} \land \textit{ThreadCreateDtosState}) \\
&\quad \gg \textit{ThreadCreateReply}
\end{aligned}$$

An unsuccessful request returns an error status.

$$\begin{aligned}
&\textit{ThreadCreateBad} \\
&\quad \mathrel{\widehat=} \textit{RVThreadCreateResourceShortage} \gg \textit{RequestNoOp} \\
&\textit{ThreadCreateSecureBad} \\
&\quad \mathrel{\widehat=} (\textit{RVThreadCreateSecureResourceShortage} \\
&\qquad \lor \textit{RVThreadCreateSecureInsufficientPermission}) \\
&\quad \gg \textit{RequestNoOp}
\end{aligned}$$

Execution of the request consists of a good execution or an error execution.

$$\begin{aligned}
&\textit{ExecuteThreadCreate} \mathrel{\widehat=} (\textit{ThreadCreateGood} \lor \textit{ThreadCreateBad}) \setminus (\textit{port}, \textit{thread}) \\
&\textit{ExecuteThreadCreateSecure} \\
&\quad \mathrel{\widehat=} (\textit{ThreadCreateSecureGood} \lor \textit{ThreadCreateSecureBad}) \setminus (\textit{port}, \textit{thread})
\end{aligned}$$

The full specification for kernel processing of a validated **thread_create** or **thread_create_secure** request consists of processing the request followed by its execution.

$$\begin{aligned}
&\textit{ThreadCreate} \mathrel{\widehat=} \textit{ProcessingThreadCreate} \mathbin{⨾} \textit{ExecuteThreadCreate} \\
&\textit{ThreadCreateSecure} \mathrel{\widehat=} \textit{ProcessingThreadCreateSecure} \mathbin{⨾} \textit{ExecuteThreadCreateSecure}
\end{aligned}$$

9.4 thread_depress_abort

The request **thread_depress_abort** restores the original scheduling priority to a thread whose priority has been set to the lowest possible value by a **swtch_pri** or **thread_switch** request.

9.4.1 Client Interface

```
kern_return_t thread_depress_abort
        (mach_port_t                thread_name);
```

9.4.1.1 Input Parameters The following input parameters are provided by the client of a **thread_depress_abort** request:

- $thread\_name?$ — the client's name for the thread whose priority depression will be canceled

$$\text{ThreadDepressAbortClientInputs}$$
$$thread\_name? : NAME$$

A **thread_depress_abort** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_depress\_abort\_id$ and has no body.

$$\text{InvokeThreadDepressAbort}$$
$$InvokeMachMsg$$
$$ThreadDepressAbortClientInputs$$
$$name? = thread\_name?$$
$$operation? = Thread\_depress\_abort\_id$$

9.4.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_depress_abort** request:

- $return!$ — the status of the request

$$\text{ThreadDepressAbortClientOutputs}$$
$$return! : KERNEL\_RETURN$$

$$\text{ThreadDepressAbortReceiveReply}$$
$$InvokeMachMsgRcv$$
$$ThreadDepressAbortClientOutputs$$
$$return! = Text\_to\_status(msg\_body)$$

9.4.2 Kernel Interface

9.4.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_depress_abort** request:

- $thread?$ — the thread whose priority depression will be canceled

$$\text{ThreadDepressAbortInputs}$$
$$thread? : THREAD$$

9.4.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_depress_abort** request:

- $return!$ — the status of the request

$$\begin{array}{|l} \hline \textit{ThreadDepressAbortOutputs} \\ return! : KERNEL\_RETURN \\ \hline \end{array}$$

9.4.3 Request Criteria

No criteria are defined for the **thread_depress_abort** request.

9.4.4 Return Values

Table 20 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$
$Kern\_success$

Table 20: Return Values for **thread_depress_abort**

$$\begin{array}{|l} \hline \textit{RVThreadDepressAbortGood} \\ \textit{ThreadDepressAbortOutputs} \\ \hline return! = Kern\_success \\ \hline \end{array}$$

9.4.5 State Changes

A successful **thread_depress_abort** request returns the priority of the thread to its value before the depression. If the priority of the thread is not currently depressed, no changes occur. Note that the scheduling priority may also change, but since we do not have enough detail in our model to compute its value we will leave it unspecified.

$$
\begin{array}{l}
\textit{ThreadDepressAbortState} \\
\hline
\Delta\ ThreadPri \\
\Delta\ Threads \\
\Xi\ ThreadExist \\
\Xi\ PortExist \\
\Xi\ TasksAndThreads \\
\Xi\ ThreadSchedPolicy \\
\Xi\ ThreadInstruction \\
\Xi\ ThreadMachineState \\
\Delta\ ThreadStatistics \\
\Delta\ Events \\
\Xi\ PortNameSpace \\
\Xi\ SpecialPurposePorts \\
\Xi\ ThreadAndProcessorSet \\
ThreadInvariants \\
thread? : THREAD \\
\hline
thread? \in \operatorname{dom} \underline{p}riority\_before\_depression \\
\\
\underline{t}hread\_priority' = \underline{t}hread\_priority \\
\quad \oplus \{thread? \mapsto \underline{p}riority\_before\_depression(thread?)\} \\
\underline{t}hread\_max\_priority' = \underline{t}hread\_max\_priority \\
\underline{d}epressed\_threads' = \underline{d}epressed\_threads \setminus \{thread?\} \\
\underline{p}riority\_before\_depression' = \underline{p}riority\_before\_depression \\
\hline
\end{array}
$$

9.4.6 Complete Request

The following schemas define the general form of a **thread_depress_abort** request.

$$
\begin{array}{l}
\textit{ProcessingThreadDepressAbort} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_depress\_abort\_id \\
\hline
\end{array}
$$

A request makes the state changes described in the previous section.

$$
\begin{array}{l}
ThreadDepressAbortGood \mathrel{\widehat=} (RVThreadDepressAbortGood \land ThreadDepressAbortState) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

Review Note:
This definition is included only for consistency with other request specifications.

Execution of the request consists of a good execution.

$$ExecuteThreadDepressAbort \mathrel{\widehat=} ThreadDepressAbortGood$$

The full specification for kernel processing of a validated **thread_depress_abort** request consists of processing the request followed by its execution.

$$ThreadDepressAbort \mathrel{\widehat=} ProcessingThreadDepressAbort \mathbin{⨾} ExecuteThreadDepressAbort$$

9.5 thread_disable_pc_sampling

The request **thread_disable_pc_sampling** turns off all sampling for a thread.

9.5.1 Client Interface

```
kern_return_t thread_disable_pc_sampling
              (mach_port_t                    thread_name,
               int                            *sample_cnt);
```

Review Note:
The DTOS KID incorrectly includes *flavor* as an input parameter of **thread_disable_pc_sampling**. This parameter is not present in the prototype. The request disables sampling of all types, not for just a particular type, and therefore there is no need for a *flavor* parameter.

9.5.1.1 Input Parameters The following input parameters are provided by the client of a **thread_disable_pc_sampling** request:

- *thread_name*? — the client's name for the thread for which sampling will be turned off

ThreadDisablePCSamplingClientInputs
$thread\_name? : NAME$

A **thread_disable_pc_sampling** request is invoked by sending a message to the port indicated by *thread_name*? that has the operation field set to *Thread_disable_pc_sampling_id* and has no body.

InvokeThreadDisablePCSampling
InvokeMachMsg
ThreadDisablePCSamplingClientInputs

$name? = thread\_name?$
$operation? = Thread\_disable\_pc\_sampling\_id$

9.5.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_disable_pc_sampling** request:

- *return*! — the status of the request
- *sample_cnt*! — the number of sample elements in the kernel for the thread

---

Editorial Note:
In the prototype this parameter is present, but unused. Thus, its output value will be whatever the input value is. The parameter should probably not be present at all since with the current semantics of thread sampling all samples are discarded when sampling is disabled for a thread. To reflect this we will define this value to be zero.

$ThreadDisablePCSampling\,ClientOutputs$
$return! : KERNEL\_RETURN$
$sample\_cnt! : \mathbb{N}$

$ThreadDisablePCSampling\,ReceiveReply$
$Invoke\,MachMsgRcv$
$ThreadDisablePCSampling\,ClientOutputs$

$(sample\_cnt!, return!) = Text\_to\_count\_and\_status(msg\_body)$

9.5.2 Kernel Interface

9.5.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_disable_pc_sampling** request:

- $thread?$ — the thread for which sampling will be turned off

$ThreadDisablePCSampling\,Inputs$
$thread? : THREAD$

9.5.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_disable_pc_sampling** request:

- $return!$ — the status of the request
- $sample\_cnt!$ — the number of sample elements in the kernel for the thread

$ThreadDisablePCSampling\,Outputs$
$return! : KERNEL\_RETURN$
$sample\_cnt! : \mathbb{N}$

Upon completion of the processing of a **thread_disable_pc_sampling** request a reply message is built from the output parameters.

$ThreadDisablePCSampling\,Reply$
$RequestReturn$
$sample\_cnt? : \mathbb{N}$

$reply? = Return\_sample\_cnt(sample\_cnt?)$

9.5.3 Request Criteria

No criteria are defined for the **thread_disable_pc_sampling** request.

$sample\_cnt!$	$return!$
0	$Kern\_success$

Table 21: Return Values for **thread_disable_pc_sampling**

9.5.4 Return Values

Table 21 describes the values returned at the completion of the request and the conditions under which each value is returned.

$$
\begin{array}{l}
\textit{RVThreadDisablePCSamplingGood} \\
\hline
ThreadDisablePCSamplingOutputs \\
\hline
sample\_cnt! = 0 \\
return! = Kern\_success \\
\hline
\end{array}
$$

9.5.5 State Changes

A successful **thread_disable_pc_sampling** request removes the thread from the set of sampled threads and from the domains of the functions describing sampling. All samples are discarded.

$$
\begin{array}{l}
\textit{ThreadDisablePCSamplingState} \\
\hline
\Delta\ Threads \\
\Delta\ ThreadSampling \\
\Xi\ ThreadPri \\
\Xi\ TasksAndThreads \\
\Xi\ ThreadSchedPolicy \\
\Xi\ ThreadInstruction \\
\Xi\ ThreadMachineState \\
\Xi\ ThreadExecStatus \\
\Xi\ Events \\
\Xi\ ThreadExist \\
\Xi\ ThreadAndProcessorSet \\
\Xi\ PortExist \\
\Xi\ PortNameSpace \\
\Xi\ SpecialPurposePorts \\
ThreadInvariants \\
thread? : THREAD \\
\hline
\underline{s}ampled\_threads' = \underline{s}ampled\_threads \setminus \{thread?\} \\
\underline{t}hread\_sample\_types' = \{thread?\} \lhd \underline{t}hread\_sample\_types \\
\underline{t}hread\_sample\_sequence\_number' = \{thread?\} \lhd \underline{t}hread\_sample\_sequence\_number \\
\underline{t}hread\_samples' = \{thread?\} \lhd \underline{t}hread\_samples \\
\hline
\end{array}
$$

9.5.6 Complete Request

The following schemas define the general form of a **thread_disable_pc_sampling** request.

$ProcessingThreadDisablePCSampling$

$ProcessThreadViaThreadPortRequestGood$

$operation? = Thread\_disable\_pc\_sampling\_id$

A request makes the state changes described in the previous section and creates a kernel reply.

$$ThreadDisablePCSamplingGood \\ \quad \widehat{=} (RVThreadDisablePCSamplingGood \land ThreadDisablePCSamplingState) \\ \quad\quad \gg ThreadDisablePCSamplingReply$$

Execution of the request consists of a good execution.

$$ExecuteThreadDisablePCSampling \widehat{=} ThreadDisablePCSamplingGood$$

The full specification for kernel processing of a validated **thread_disable_pc_sampling** request consists of processing the request followed by its execution.

$$ThreadDisablePCSampling \widehat{=} ProcessingThreadDisablePCSampling \\ \quad ⨾ ExecuteThreadDisablePCSampling$$

9.6 thread_enable_pc_sampling

The request **thread_enable_pc_sampling** turns on a given type of sampling for a thread.

9.6.1 Client Interface

```
kern_return_t thread_enable_pc_sampling
                (mach_port_t              thread_name,
                 int                      *ticks,
                 sampled_pc_flavor_t      flavor);
```

9.6.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_enable_pc_sampling** request:

- $thread\_name?$ — the client's name for the thread for which sampling will be turned on
- $flavor?$ — the type of samples to collect

$ThreadEnablePCSamplingClientInputs$

$thread\_name? : NAME$
$flavor? : \mathbb{P}\, SAMPLE\_TYPES$

A **thread_enable_pc_sampling** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_enable\_pc\_sampling\_id$ and has a body consisting of $flavor?$.

$$\begin{array}{|l}
\hline
\textit{InvokeThreadEnablePCSampling} \\
\textit{InvokeMachMsg} \\
\textit{ThreadEnablePCSamplingClientInputs} \\
\hline
name? = thread_name? \\
operation? = Thread_enable_pc_sampling_id \\
msg_body = Sample_type_set_to_text(flavor?) \\
\hline
\end{array}$$

9.6.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_enable_pc_sampling** request:

- $return!$ — the status of the request
- $ticks!$ — the clock granularity (ticks per second) according to the kernel

$$\begin{array}{|l}
\hline
\textit{ThreadEnablePCSamplingClientOutputs} \\
return! : KERNEL_RETURN \\
ticks! : \mathbb{N}_1 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{ThreadEnablePCSamplingReceiveReply} \\
\textit{InvokeMachMsgRcv} \\
\textit{ThreadEnablePCSamplingClientOutputs} \\
\hline
(ticks!, return!) = Text_to_ticks_and_status(msg_body) \\
\hline
\end{array}$$

9.6.2 Kernel Interface

9.6.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_enable_pc_sampling** request:

- $thread?$ — the thread for which sampling will be turned on
- $flavor?$ — the type of samples to collect

$$\begin{array}{|l}
\hline
\textit{ThreadEnablePCSamplingInputs} \\
thread? : THREAD \\
flavor? : \mathbb{P}\, SAMPLE_TYPES \\
\hline
\end{array}$$

9.6.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_enable_pc_sampling** request:

- $return!$ — the status of the request
- $ticks!$ — the clock granularity (ticks per second) according to the kernel

$$\begin{array}{|l} \hline \textit{ThreadEnablePCSamplingOutputs} \\ return! : KERNEL\_RETURN \\ ticks! : \mathbb{N}_1 \\ \hline \end{array}$$

Upon completion of the processing of a **thread_enable_pc_sampling** request a reply message is built from the output parameters.

$$\begin{array}{|l} \hline \textit{ThreadEnablePCSamplingReply} \\ RequestReturn \\ ticks? : \mathbb{N}_1 \\ \hline reply? = Return\_ticks(ticks?) \\ \hline \end{array}$$

9.6.3 Request Criteria

The following criteria are defined for the **thread_enable_pc_sampling** request.

- **C1 —** There are sufficient resources to create a sampling buffer. We do not actually model the consumption of resources by the kernel. So, we will use the set $Resources\_available\_to\_create\_sampling\_buffer$ to indicate the set of states where there are sufficient resources to create a sampling buffer.

$$| \; Resources\_available\_to\_create\_sampling\_buffer : \mathbb{P}\, DtosExec$$

$$\begin{array}{|l} \hline \textit{C1ThreadEnablePCSamplingResourcesAvailable} \\ DtosExec \\ \hline \theta DtosExec \in Resources\_available\_to\_create\_sampling\_buffer \\ \hline \end{array}$$

$$NotC1ThreadEnablePCSamplingResourcesAvailable \mathrel{\widehat=} \\ \quad DtosExec \wedge \neg\ C1ThreadEnablePCSamplingResourcesAvailable$$

Note that no criterion is defined to check that $flavor?$ is a set of recognized sample types. If an unrecognized type is included in $flavor?$, no error will occur. Unrecognized sample types will simply be ignored and produce no samples.

9.6.4 Return Values

Table 22 describes the values returned at the completion of the request and the conditions under which each value is returned. The design does not specify the value of $ticks!$ when an error occurs. It depends on the implementation, and we leave it unspecified.

$$| \; Ticks\_per\_second : \mathbb{N}_1$$

Editorial Note:
Even though C1 examines resource availability, the kernel returns $Kern\_invalid\_argument$ when C1 is false. In addition the following message is printed to standard output: "thread_enable_pc_sampling: kalloc failed".

$ticks!$	$return!$	C1
$Ticks\_per\_second$	$Kern\_success$	T
—	$Kern\_invalid\_argument$	F

Table 22: Return Values for **thread_enable_pc_sampling**

$$
\begin{array}{|l}
\hline
RVThreadEnablePCSamplingGood \\
C1\,ThreadEnablePCSamplingResourcesAvailable \\
ThreadEnablePCSampling\,Outputs \\
\hline
ticks! = Ticks\_per\_second \\
return! = Kern\_success \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
RVThreadEnablePCSamplingResourceShortage \\
NotC1\,ThreadEnablePCSamplingResourcesAvailable \\
ThreadEnablePCSampling\,Outputs \\
\hline
return! = Kern\_invalid\_argument \\
\hline
\end{array}
$$

9.6.5 State Changes

A successful **thread_enable_pc_sampling** request adds the thread to the set of sampled threads and records the type of samples to be collected. It also sets the sample sequence number for the thread to zero. If the thread was already being sampled, the flavor is reset, but the sequence number is unchanged. In this case, any samples currently in the buffer remain there.

$$
\begin{array}{l}
\textit{ThreadEnablePCSamplingState} \\
\hline
\Delta\ Threads \\
\Delta\ ThreadSampling \\
\Xi\ ThreadPri \\
\Xi\ TasksAndThreads \\
\Xi\ ThreadSchedPolicy \\
\Xi\ ThreadInstruction \\
\Xi\ ThreadMachineState \\
\Xi\ ThreadExecStatus \\
\Xi\ Events \\
\Xi\ ThreadExist \\
\Xi\ ThreadAndProcessorSet \\
\Xi\ PortExist \\
\Xi\ PortNameSpace \\
\Xi\ SpecialPurposePorts \\
ThreadInvariants \\
thread? : THREAD \\
flavor? : \mathbb{P}\ SAMPLE\_TYPES \\
\hline
\underline{s}ampled\_threads' = \underline{s}ampled\_threads \cup \{thread?\} \\
\underline{t}hread\_sample\_types' = \underline{t}hread\_sample\_types \oplus \{thread? \mapsto flavor?\} \\
\underline{t}hread\_sample\_sequence\_number' = \{thread? \mapsto 0\} \\
\quad \oplus\ \underline{t}hread\_sample\_sequence\_number \\
\underline{t}hread\_samples' = \{thread? \mapsto \langle\rangle\} \oplus \underline{t}hread\_samples
\end{array}
$$

9.6.6 Complete Request

The following schemas define the general form of a **thread_enable_pc_sampling** request.

$$
\begin{array}{l}
\textit{ProcessingThreadEnablePCSampling} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_enable\_pc\_sampling\_id
\end{array}
$$

A request makes the state changes described in the previous section and creates a kernel reply.

$$
\begin{array}{l}
ThreadEnablePCSamplingGood \\
\quad \widehat{=}\ (RVThreadEnablePCSamplingGood \land ThreadEnablePCSamplingState) \\
\qquad \gg\ ThreadEnablePCSamplingReply
\end{array}
$$

An unsuccessful request returns an error status.

$$
\begin{array}{l}
ThreadEnablePCSamplingBad \\
\quad \widehat{=}\ RVThreadEnablePCSamplingResourceShortage \gg RequestNoOp
\end{array}
$$

Execution of the request consists of a good execution or an error execution.

$$
\begin{array}{l}
ExecuteThreadEnablePCSampling \ \widehat{=}\ ThreadEnablePCSamplingGood \\
\quad \land\ ThreadEnablePCSamplingBad
\end{array}
$$

The full specification for kernel processing of a validated **thread_enable_pc_sampling** request consists of processing the request followed by its execution.

$$ThreadEnablePCSampling \mathrel{\widehat=} Processing\,ThreadEnablePCSampling \\ \quad ⨾\; Execute\,ThreadEnablePCSampling$$

9.7 thread_get_assignment

The request **thread_get_assignment** returns a send right to the name port of the processor set to which a thread is assigned. This port can only be used to obtain information about the processor set.

9.7.1 Client Interface

```
kern_return_t thread_get_assignment
              (mach_port_t          thread_name,
               mach_port_t*         processor_set_name);
```

9.7.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_get_assignment** request:

- $thread\_name?$ — the client's name for the thread whose processor set name port is requested

$$\begin{array}{|l}
\hline ThreadGetAssignment\,ClientInputs \\
thread_name? : NAME \\
\hline
\end{array}$$

A **thread_get_assignment** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_get\_assignment\_id$ and has no body.

$$\begin{array}{|l}
\hline Invoke\,ThreadGetAssignment \\
Invoke\,MachMsg \\
ThreadGetAssignment\,ClientInputs \\
\hline
name? = thread_name? \\
operation? = Thread_get_assignment_id \\
\hline
\end{array}$$

9.7.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_get_assignment** request:

- $processor\_set\_name!$ — a send right to the name port of the desired processor set
- $return!$ — the status of the request

$$\begin{array}{|l}
\hline ThreadGetAssignment\,ClientOutputs \\
processor_set_name! : NAME \\
return! : KERNEL_RETURN \\
\hline
\end{array}$$

$$
\begin{array}{|l}
\hline
\textit{ThreadGetAssignmentReceiveReply} \\
\textit{InvokeMachMsgRcv} \\
\textit{ThreadGetAssignmentClientOutputs} \\
\hline
(processor\_set\_name!, return!) = Text\_to\_name\_and\_status(msg\_body) \\
\hline
\end{array}
$$

9.7.2 Kernel Interface

9.7.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_get_assignment** request:

- $thread?$ — the thread whose processor set name port is requested

$$
\begin{array}{|l}
\hline
\textit{ThreadGetAssignmentInputs} \\
thread? : THREAD \\
\hline
\end{array}
$$

9.7.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_get_assignment** request:

- $processor\_set!$ — the desired processor set
- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{ThreadGetAssignmentOutputs} \\
processor\_set! : PROCESSOR\_SET \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

Upon completion of the processing of a **thread_get_assignment** request a reply message is built from the output parameters.

$$
\begin{array}{|l}
\hline
\textit{ThreadGetAssignmentReply} \\
\textit{RequestOnlyObserves} \\
processor\_set? : PROCESSOR\_SET \\
\hline
\textbf{let } port == \underline{ps}\_name\_port\_rel(processor\_set?) \\
\bullet\ reply? = Return\_capability(Thread\_port\_to\_s\_right(port)) \\
\hline
\end{array}
$$

9.7.3 Request Criteria

No criteria are defined for the **thread_get_assignment** request.

9.7.4 Return Values

Table 23 describes the values returned at the completion of the request and the conditions under which each value is returned.

$processor\_set!$	$return!$
$thread\_assigned\_to(thread?)$	$Kern\_success$

Table 23: Return Values for **thread_get_assignment**

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RVThreadGetAssignmentGood}} \\
\hline
ThreadAndProcessorSet \\
ProcessorsAndPorts \\
ThreadGetAssignmentOutputs \\
thread? : THREAD \\
\hline
thread? \in \mathrm{dom}\ thread\_assigned\_to \\
\\
return! = Kern\_success \\
processor\_set! = thread\_assigned\_to(thread?) \\
\hline
\end{array}
$$

9.7.5 State Changes

A **thread_get_assignment** request does not make any state changes since it only observes the system state.

9.7.6 Complete Request

The **thread_get_assignment** request has the following general form.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProcessingThreadGetAssignment}} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_get\_assignment\_id \\
\hline
\end{array}
$$

The full specification for kernel processing of a validated **thread_get_assignment** request consists of processing the request, execution of the request, and the creation of a kernel reply.

$$
\begin{array}{l}
ThreadGetAssignment \mathrel{\widehat=} ProcessingThreadGetAssignment \\
\quad \fatsemi (RVThreadGetAssignmentGood \gg ThreadGetAssignmentReply)
\end{array}
$$

9.8 thread_get_sampled_pcs

The request **thread_get_sampled_pcs** returns the samples collected for a given thread.

9.8.1 Client Interface

```
kern_return_t thread_get_sampled_pcs
              (mach_port_t        thread_name,
               unsigned           *seqno,
               sampled_pc_t       sampled_pcs[ ],
               int                *sample_cnt);
```

9.8.1.1 Input Parameters The following input parameters are provided by the client of a **thread_get_sampled_pcs** request:

- $thread\_name?$ — the client's name for the thread whose samples will be returned
- $seqno?$ — the sequence number of the first sample that should be returned. If this sample is no longer available due to insufficient space in the sampling buffer, the earliest available sample will be the starting point.

Review Note:
The DTOS KID reports $seqno$ as an output parameter only. In the prototype it is used for both input and output.

$ThreadGetSampledPCsClientInputs$
$thread\_name? : NAME$
$seqno? : \mathbb{N}$

A **thread_get_sampled_pcs** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_get\_sampled\_pcs\_id$ and has a body consisting of $seqno?$.

$InvokeThreadGetSampledPCs$
$InvokeMachMsg$
$ThreadGetSampledPCsClientInputs$

$name? = thread\_name?$
$operation? = Thread\_get\_sampled\_pcs\_id$
$msg\_body = Sequence\_number\_to\_text(seqno?)$

9.8.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_get_sampled_pcs** request:

- $return!$ — the status of the request
- $seqno!$ — the sequence number of the most recently collected sample
- $sampled\_pcs!$ — the samples returned
- $sample\_cnt!$ — the number of samples returned

$ThreadGetSampledPCsClientOutputs$
$return! : KERNEL\_RETURN$
$seqno! : \mathbb{N}$
$sampled\_pcs! : \mathrm{seq}\, SAMPLE$
$sample\_cnt! : \mathbb{Z}$

$ThreadGetSampledPCsReceiveReply$
$InvokeMachMsgRcv$
$ThreadGetSampledPCsClientOutputs$

$(seqno!, sampled\_pcs!, sample\_cnt!, return!)$
$= Text\_to\_seqno\_and\_PCs\_and\_count\_and\_status(msg\_body)$

9.8.2 Kernel Interface

9.8.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_get_sampled_pcs** request:

- $thread?$ — the thread whose samples will be returned
- $seqno?$ — the sequence number of the first sample that should be returned. If this sample is no longer available due to insufficient space in the sampling buffer, the earliest available sample will be the starting point.

> *Review Note:*
> The DTOS KID reports $seqno$ as an output parameter only. In the prototype it is used for both input and output.

$ThreadGetSampledPCsInputs$
$thread? : THREAD$
$seqno? : \mathbb{N}$

9.8.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_get_sampled_pcs** request:

- $return!$ — the status of the request
- $seqno!$ — the sequence number of the most recently collected sample
- $sampled\_pcs!$ — the samples returned
- $sample\_cnt!$ — the number of samples returned

$ThreadGetSampledPCsOutputs$
$return! : KERNEL\_RETURN$
$seqno! : \mathbb{N}$
$sampled\_pcs! : \mathrm{seq}\, SAMPLE$
$sample\_cnt! : \mathbb{Z}$

Upon completion of the processing of a **thread_get_sampled_pcs** request a reply message is built from the output parameters.

$ThreadGetSampledPCsReply$
$RequestOnlyObserves$
$seqno? : \mathbb{N}$
$sampled\_pcs? : \mathrm{seq}\, SAMPLE$
$sample\_cnt? : \mathbb{Z}$

$reply? = Return\_samples(seqno?, sampled\_pcs?, sample\_cnt?)$

9.8.3 Request Criteria

The following criteria are defined for the **thread_get_sampled_pcs** request.

- **C1 —** Sampling is currently enabled for the thread.

$$
\begin{array}{l}
\textbf{schema } C1ThreadSamplingEnabled \\
\quad ThreadSampling \\
\quad thread? : THREAD \\
\hline
\quad thread? \in \underline{s}ampled\_threads
\end{array}
$$

$$NotC1ThreadSamplingEnabled \mathrel{\widehat=} ThreadSampling \land \lnot\ C1ThreadSamplingEnabled$$

9.8.4 Return Values

Tables 24–27 describe the values returned at the completion of the request and the conditions under which each value is returned. The specification does not state what should be returned in $seqno!$, $sampled\_pcs!$ and $sample\_cnt!$ when the thread is not being sampled. So, these values depend on the implementation and we leave them unspecified.

$seqno!$	C1
$\underline{t}hread\_sample\_sequence\_number(thread?)$	T
—	F

Table 24: Return Values for **thread_get_sampled_pcs**

$sampled\_pcs!$	C1
$Samples\_returned(\underline{t}hread\_samples(thread?), seqno?, \underline{t}hread\_sample\_sequence\_number(thread?))$	T
—	F

Table 25: Return Values for **thread_get_sampled_pcs**

$sample\_cnt!$	C1
$\# sampled\_pcs!$	T
—	F

Table 26: Return Values for **thread_get_sampled_pcs**

The function $Samples\_returned\_count(from, to)$ returns the number of samples in the range $from$ to to or the buffer size $Max\_samples$, whichever is smaller. A negative return value is interpreted as zero samples. The function $Samples\_returned(sample\_sequence, from, to)$ returns a number of samples as indicated by $Samples\_returned\_count(from, to)$ from $sample\_sequence$ ending with the sample with sequence number to.

$return!$	C1
$Kern\_success$	T
$Kern\_failure$	F

Table 27: Return Values for **thread_get_sampled_pcs**

$$
\begin{array}{l}
Samples\_returned : (\mathrm{seq}\, SAMPLE) \times \mathbb{N} \times \mathbb{N} \pfun \mathrm{seq}\, SAMPLE \\
Samples\_returned\_count : (\mathbb{N} \times \mathbb{N}) \pfun \mathbb{Z} \\
\hline
\forall\, sample\_sequence : \mathrm{seq}\, SAMPLE; from, to : \mathbb{N} \\
\bullet\ Samples\_returned\_count(from, to) = min\, \{Max\_samples, to - from + 1\} \\
\land\ Samples\_returned(sample\_sequence, from, to) \\
\qquad = \{j : \mathbb{N} \mid to - Samples\_returned\_count(from, to) < j \leq to\} \upharpoonright sample\_sequence
\end{array}
$$

$$
\begin{array}{l}
\textit{RVThreadGetSampledPCsGood} \\
\hline
C1\,ThreadSamplingEnabled \\
ThreadGetSampledPCs\,Outputs \\
ThreadGetSampledPCs\,Inputs \\
\hline
seqno! = \underline{t}hread\_sample\_sequence\_number(thread?) \\
sampled\_pcs! = Samples\_returned(\underline{t}hread\_samples(thread?), \\
\qquad seqno?, \underline{t}hread\_sample\_sequence\_number(thread?)) \\
sample\_cnt! = \#sampled\_pcs! \\
return! = Kern\_success
\end{array}
$$

$$
\begin{array}{l}
\textit{RVThreadGetSampledPCsBad} \\
\hline
NotC1\,ThreadSamplingEnabled \\
ThreadGetSampledPCs\,Outputs \\
ThreadGetSampledPCs\,Inputs \\
\hline
return! = Kern\_failure
\end{array}
$$

9.8.5 State Changes

A **thread_get_sampled_pcs** request does not make any state changes since it only observes the system state.

9.8.6 Complete Request

The following schemas define the general form of a **thread_get_sampled_pcs** request.

$$
\begin{array}{l}
\textit{Processing\,ThreadGetSampledPCs} \\
\hline
ProcessThread\,Via\,ThreadPortRequestGood \\
\hline
operation? = Thread\_get\_sampled\_pcs\_id
\end{array}
$$

A successful request creates a kernel reply.

$$
\begin{array}{l}
ThreadGetSampledPCs\,Good \mathrel{\widehat=} RVThreadGetSampledPCsGood \\
\qquad \gg\ ThreadGetSampledPCs\,Reply
\end{array}
$$

An unsuccessful request returns an error status.

$$ThreadGetSampledPCsBad \mathrel{\widehat=} RVThreadGetSampledPCsBad \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadGetSampledPCs \mathrel{\widehat=} ThreadGetSampledPCsGood \lor ThreadGetSampledPCsBad$$

The full specification for kernel processing of a validated **thread_get_sampled_pcs** request consists of processing the request followed by its execution.

$$ThreadGetSampledPCs \mathrel{\widehat=} ProcessingThreadGetSampledPCs \\ \quad ⨾ \; ExecuteThreadGetSampledPCs$$

9.9 thread_get_special_port

The request **thread_get_special_port** allows a task to obtain a send right to a specified special port for a specified thread.

9.9.1 Client Interface

kern_return_t **thread_get_special_port**
(mach_port_t *thread_name*,
int *which_port*,
mach_port_t* *special_port_name*);

thread_get_exception_port
Macro form

kern_return_t **thread_get_exception_port**
(mach_port_t *thread_name*,
mach_port_t* *special_port_name*);

⇒ **thread_get_special_port** (*thread_name*, THREAD_EXCEPTION_PORT,
special_port_name)

thread_get_kernel_port
Macro form

kern_return_t **thread_get_kernel_port**
(mach_port_t *thread_name*,
mach_port_t* *special_port_name*);

⇒ **thread_get_special_port** (*thread_name*, THREAD_KERNEL_PORT,
special_port_name)

9.9.1.1 Input Parameters The following input parameters are provided by the client of a **thread_get_special_port** request:

- $thread\_name?$ — the client's name for the thread whose special port is to be returned
- $which\_port?$ — the type of special port that is to be returned

```
┌─ ThreadGetSpecialPortClientInputs ─────────────────
│ thread_name? : NAME
│ which_port? : THREAD_SPECIAL_PORTS
└────────────────────────────────────────────────────
```

A **thread_get_special_port** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_get\_special\_port\_id$ and has a body consisting of $which\_port?$.

```
┌─ InvokeThreadGetSpecialPort ───────────────────────
│ Invoke
│ ThreadGetSpecialPortClientInputs
├──────────
│ trap_id? = Mach_msg_trap
│ user_spec?.message.header.operation = Thread_get_special_port_id
│ user_spec?.message.header.remote_port = thread_name?
│ user_spec?.message.body
│       = Thread_special_ports_to_text(which_port?)
└────────────────────────────────────────────────────
```

9.9.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_get_special_port** request:

- $special\_port\_name!$ — the name of a send right (capability) for the requested special port
- $return!$ — the status of the request

```
┌─ ThreadGetSpecialPortClientOutputs ────────────────
│ special_port_name! : NAME
│ return! : KERNEL_RETURN
└────────────────────────────────────────────────────
```

```
┌─ ThreadGetSpecialPortReceiveReply ─────────────────
│ InvokeMachMsgRcv
│ ThreadGetSpecialPortClientOutputs
├──────────
│ (special_port_name!, return!) = Text_to_name_and_status(msg_body)
└────────────────────────────────────────────────────
```

9.9.2 Kernel Interface

9.9.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_get_special_port** request:

- $thread?$ — the thread whose special port is to be returned

- $which\_port?$ — the type of special port that is to be returned

$$
\begin{array}{|l}
\hline
\textit{ThreadGetSpecialPortInputs} \\
thread? : THREAD \\
which\_port? : THREAD\_SPECIAL\_PORTS \\
\hline
\end{array}
$$

9.9.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_get_special_port** request:

- $special\_port!$ — the requested special port
- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{ThreadGetSpecialPortOutputs} \\
special\_port! : PORT \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

Upon completion of the processing of a **thread_get_special_port** request a reply message is built from the output parameters. The reply message will contain a send right to the requested special port.

$$
\begin{array}{|l}
\hline
\textit{ThreadGetSpecialPortReply} \\
OnlyObserves \\
special\_port? : PORT \\
\hline
reply? = Return\_capability(Thread\_port\_to\_s\_right(special\_port?)) \\
\hline
\end{array}
$$

9.9.3 Request Criteria

The following criteria are defined for the **thread_get_special_port** request.

- **C1 —** An exception port request is made.

$$
\begin{array}{|l}
\hline
C1\,ThreadGetExceptionPort \\
which\_port? : THREAD\_SPECIAL\_PORTS \\
\hline
which\_port? = Thread\_exception\_port \\
\hline
\end{array}
$$

$$NotC1\,ThreadGetExceptionPort \mathrel{\widehat{=}} \neg\ C1\,ThreadGetExceptionPort$$

- **C2 —** A kernel port request is made.

$$
\begin{array}{|l}
\hline
C2\,ThreadGetKernelPort \\
which\_port? : THREAD\_SPECIAL\_PORTS \\
\hline
which\_port? = Thread\_kernel\_port \\
\hline
\end{array}
$$

$$NotC2\,ThreadGetKernelPort \mathrel{\widehat{=}} \neg\ C2\,ThreadGetKernelPort$$

- **C3 —** The client has $Get\_thread\_exception\_port$ permission to the target thread.

Review Note:
In C3 and C4, we've begun the process of dealing with deferred permission checks under the new execution model. The first schema is used to initiate the permission checking routine and the criteria schemas will be used after the permission has been retrieved.

$$
\begin{array}{l}
\textbf{ThreadGetSpecialPortPermCheckGTEP} \\
Transition \\
\hline
\exists\, request : Request;\ CheckPending;\ ThreadGetSpecialPortInputs \\
\bullet\ curr\_bk?? = Bk\_have\_request(request) \\
\quad \land\ request.operation = Thread\_get\_special\_port\_id \\
\quad \land\ thread\_self(thread?) = request.service\_port \\
\quad \land\ ssi = \underline{t}ask\_sid(curr\_task??) \\
\quad \land\ osi = thread\_target(curr\_task??, thread?) \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Get\_thread\_exception\_port, env) \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{C3ThreadCanGetExceptionPort} \\
Transition \\
env : ENVIRONMENT \\
\hline
curr\_bk?? = Bk\_have\_ruling(Get\_thread\_exception\_port, True, env)
\end{array}
$$

$$
\begin{array}{l}
\textbf{NotC3ThreadCanGetExceptionPort} \\
Transition \\
env : ENVIRONMENT \\
\hline
curr\_bk?? = Bk\_have\_ruling(Get\_thread\_exception\_port, False, env)
\end{array}
$$

- **C4 —** The client has $Get\_thread\_kernel\_port$ permission to the target thread.

$$
\begin{array}{l}
\textbf{ThreadGetSpecialPortPermCheckGTKP} \\
Transition \\
\hline
\exists\, request : Request;\ CheckPending;\ ThreadGetSpecialPortInputs \\
\bullet\ curr\_bk?? = Bk\_have\_request(request) \\
\quad \land\ request.operation = Thread\_get\_special\_port\_id \\
\quad \land\ thread\_self(thread?) = request.service\_port \\
\quad \land\ ssi = \underline{t}ask\_sid(curr\_task??) \\
\quad \land\ osi = thread\_target(curr\_task??, thread?) \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Get\_thread\_kernel\_port, env) \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{C4ThreadCanGetKernelPort} \\
Transition \\
env : ENVIRONMENT \\
\hline
curr\_bk?? = Bk\_have\_ruling(Get\_thread\_kernel\_port, True, env)
\end{array}
$$

$$\begin{array}{l} \textit{NotC4ThreadCanGetKernelPort} \\ \hline \textit{Transition} \\ env : ENVIRONMENT \\ \hline curr\_bk?? = Bk\_have\_ruling(Get\_thread\_kernel\_port, False, env) \end{array}$$

- **C5 —** The exception port of the thread is defined.

$$\begin{array}{l} \textit{C5ThreadExceptionPortDefined} \\ \hline \textit{SpecialThreadPorts} \\ thread? : THREAD \\ \hline thread? \in \operatorname{dom} thread\_eport \end{array}$$

$$NotC5ThreadExceptionPortDefined \\ \quad \widehat{=} \; SpecialThreadPorts \land \neg \; C5ThreadExceptionPortDefined$$

- **C6 —** The kernel port of the thread is defined.

$$\begin{array}{l} \textit{C6ThreadKernelPortDefined} \\ \hline \textit{SpecialThreadPorts} \\ thread? : THREAD \\ \hline thread? \in \operatorname{dom} thread\_sself \end{array}$$

$$NotC6ThreadKernelPortDefined \\ \quad \widehat{=} \; SpecialThreadPorts \land \neg \; C6ThreadKernelPortDefined$$

9.9.4 Return Values

Tables 28 and 29 describe the values returned at the completion of the request and the conditions under which each value is returned. Note that C1 and C2 are mutually exclusive. It is possible that a thread has no exception or kernel (sself) port since the port may have been deleted. The design does not specify the value of $special\_port!$ in this case. We assume that the null port is returned by the kernel routine, and that IPC will convert this into the name $Mach\_port\_null$. We leave unspecified the value returned in $special\_port!$ when the client does not have permission to get the requested special port or when the client does not ask for a valid type of special port. Note that C5 and C6 do not affect the return status.

Review Note:
We assume that the prototype will check the conditions in the order {C1, C2 }, {C3 or C4}, {C5 or C6}. However, the prototype is currently not checking C3 and C4.

Review Note:
It might make more sense to permit $Null\_port$ in the range of $thread\_eport$ and $thread\_sself$. (Note that the exception port is actually initialized to $Null\_port$ by the **thread_create** request in the prototype and that **thread_set_special_port** can set an exception or kernel port to $Null\_port$.) This would remove the need for criteria C5 and C6.

$return!$	C1	C2	C3	C4
$Kern\_success$	T	F	T	-
$Kern\_success$	F	T	-	T
$Kern\_insufficient\_permission$	T	F	F	-
$Kern\_insufficient\_permission$	F	T	-	F
$Kern\_invalid\_argument$	F	F	-	-

Table 28: Return Values for **thread_get_special_port**

$special\_port!$	C1	C2	C3	C4	C5	C6
$thread\_eport(thread?)$	T	F	T	-	T	-
$Null\_port$	T	F	T	-	F	-
$thread\_sself(thread?)$	F	T	-	T	-	T
$Null\_port$	F	T	-	T	-	F
—	otherwise					

Table 29: Return Values for **thread_get_special_port**

$RVThreadGetExceptionPortGood$

$C1ThreadGetExceptionPort$
$NotC2ThreadGetKernelPort$
$C3ThreadCanGetExceptionPort$
$C5ThreadExceptionPortDefined$
$ThreadGetSpecialPortOutputs$
$thread? : THREAD$

$return! = Kern\_success$
$special\_port! = thread\_eport(thread?)$

$RVThreadGetExceptionPortNull$

$C1ThreadGetExceptionPort$
$NotC2ThreadGetKernelPort$
$C3ThreadCanGetExceptionPort$
$NotC5ThreadExceptionPortDefined$
$ThreadGetSpecialPortOutputs$

$return! = Kern\_success$
$special\_port! = Null\_port$

$RVThreadGetKernelPortGood$

$NotC1ThreadGetExceptionPort$
$C2ThreadGetKernelPort$
$C4ThreadCanGetKernelPort$
$C6ThreadKernelPortDefined$
$ThreadGetSpecialPortOutputs$
$thread? : THREAD$

$return! = Kern\_success$
$special\_port! = thread\_sself(thread?)$

$RVThreadGetKernelPortNull$
$NotC1ThreadGetExceptionPort$
$C2ThreadGetKernelPort$
$C4ThreadCanGetKernelPort$
$NotC6ThreadKernelPortDefined$
$ThreadGetSpecialPortOutputs$

$return! = Kern\_success$
$special\_port! = Null\_port$

$RVThreadCannotGetExceptionPort$
$C1ThreadGetExceptionPort$
$NotC2ThreadGetKernelPort$
$NotC3ThreadCanGetExceptionPort$
$ThreadGetSpecialPortOutputs$

$return! = Kern\_insufficient\_permission$

$RVThreadCannotGetKernelPort$
$NotC1ThreadGetExceptionPort$
$C2ThreadGetKernelPort$
$NotC4ThreadCanGetKernelPort$
$ThreadGetSpecialPortOutputs$

$return! = Kern\_insufficient\_permission$

$RVThreadGetSpecialPortInvalidArgument$
$NotC1ThreadGetExceptionPort$
$NotC2ThreadGetKernelPort$
$ThreadGetSpecialPortOutputs$

$return! = Kern\_invalid\_argument$

9.9.5 State Changes

A **thread_get_special_port** request does not make any state changes since it only observes the system state.

9.9.6 Complete Request

The **thread_get_special_port** request has the following general form.

$ProcessingThreadGetSpecialPort$
$ProcessThreadViaThreadPortRequestGood$

$operation? = Thread\_get\_special\_port\_id$

A successful **thread_get_special_port** request causes the creation of a kernel reply.

$$
\begin{aligned}
&ThreadGetSpecialPortGood \\
&\quad \widehat{=} (RVThreadGetExceptionPortGood \lor RVThreadGetExceptionPortNull \\
&\qquad \lor RVThreadGetKernelPortGood \lor RVThreadGetKernelPortNull) \\
&\quad \gg ThreadGetSpecialPortReply
\end{aligned}
$$

$$
\begin{aligned}
&ThreadGetSpecialPortBad \\
&\quad \widehat{=} (RVThreadCannotGetExceptionPort \lor RVThreadCannotGetKernelPort \\
&\qquad \lor RVThreadGetSpecialPortInvalidArgument) \\
&\quad \gg NoOp
\end{aligned}
$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadGetSpecialPort \mathrel{\widehat{=}} (ThreadGetSpecialPortGood \lor ThreadGetSpecialPortBad)$$

The full specification for kernel processing of a validated **thread_get_special_port** request consists of processing the request followed by its execution.

$$ThreadGetSpecialPort \mathrel{\widehat{=}} ProcessingThreadGetSpecialPort \mathbin{⨾} ExecuteThreadGetSpecialPort$$

9.10 thread_get_state

The request **thread_get_state** returns an array containing state information about a specified thread (other than the client thread).

9.10.1 Client Interface

kern_return_t **thread_get_state**		
	(mach_port_t	*target_thread_name*,
	int	*flavor*,
	thread_state_t	*old_state*,
	mach_msg_type_number_t*	*old_state_cnt*);

9.10.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_get_state** request:

- $target\_thread\_name?$ — the client's name for the thread whose state information is to be returned
- $flavor?$ — the type of state information that is to be returned
- $old\_state\_cnt?$ — the maximum size of the array to be returned

$$
\begin{array}{|l}
\hline
\textit{ThreadGetStateClientInputs} \\
target\_thread\_name? : NAME \\
flavor? : THREAD\_STATE\_INFO\_TYPES \\
old\_state\_cnt? : \mathbb{N} \\
\hline
\end{array}
$$

A **thread_get_state** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_get\_state\_id$ and has a body consisting of $flavor?$ and $old\_state\_cnt?$.

$$
\begin{array}{l}
\textit{InvokeThreadGetState} \\
\hline
\textit{InvokeMachMsg} \\
\textit{ThreadGetStateClientInputs} \\
\hline
name? = target\_thread\_name? \\
operation? = Thread\_get\_state\_id \\
msg\_body = Thread\_state\_info\_type\_and\_number\_to\_text(flavor?, old\_state\_cnt?) \\
\hline
\end{array}
$$

9.10.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_get_state** request:

- $old\_state!$ — state information of the specified type about the target thread
- $old\_state\_cnt!$ — the size of the full array of available state information of the type specified
- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadGetStateClientOutputs} \\
\hline
old\_state! : THREAD\_STATE\_INFO \\
old\_state\_cnt! : \mathbb{N} \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{ThreadGetStateReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{ThreadGetStateClientOutputs} \\
\hline
(old\_state!, old\_state\_cnt!, return!) \\
\quad = Text\_to\_state\_and\_count\_and\_status(msg\_body) \\
\hline
\end{array}
$$

9.10.2 Kernel Interface

9.10.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_get_state** request:

- $target\_thread?$ — the thread whose state information is to be returned
- $flavor?$ — the type of state information that is to be returned
- $old\_state\_cnt?$ — the maximum size of the array to be returned

$$
\begin{array}{l}
\textit{ThreadGetStateInputs} \\
\hline
target\_thread? : THREAD \\
flavor? : THREAD\_STATE\_INFO\_TYPES \\
old\_state\_cnt? : \mathbb{N} \\
\hline
\end{array}
$$

9.10.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_get_state** request:

- $old\_state!$ — state information of the specified type about the target thread
- $old\_state\_cnt!$ — the size of the full array of available state information of the type specified
- $return!$ — the status of the request

```
── ThreadGetStateOutputs ──────────────────────────
│ old_state! : THREAD_STATE_INFO
│ old_state_cnt! : ℕ
│ return! : KERNEL_RETURN
└──────────────────────────────────────────────────
```

Upon completion of the processing of a **thread_get_state** request a reply message is built from the output parameters.

```
── ThreadGetStateReply ────────────────────────────
│ RequestReturn
│ old_state? : THREAD_STATE_INFO
│ old_state_cnt? : ℕ
├──────────
│ reply? = Return_thread_state_info(old_state?, old_state_cnt?)
└──────────────────────────────────────────────────
```

9.10.3 Request Criteria

The following criteria are defined for the **thread_get_state** request.

- **C1 —** The parameter $thread?$ is not equal to the client thread, $flavor?$ is a valid type of state information, and $old\_state\_cnt?$ is large enough for the requested state information. (The function $Thread\_state\_count$ returns the size required for the given type of state information.)

Editorial Note:
Nothing in the design states the reason that the client thread may not get its own state information. We believe that it is merely an implementation difficulty in that in order to get state information the thread must be temporarily stopped. If the client thread stopped itself, it could not collect the state information.

$C1ThreadGetStateGoodArgs$

$ThreadMachineState$
$ThreadsAndProcessors$
$cpu?? : PROCESSOR$
$thread? : THREAD$
$flavor? : THREAD\_STATE\_INFO\_TYPES$
$old\_state\_cnt? : \mathbb{N}$

$cpu?? \in \mathrm{dom}\, \underline{a}ctive\_thread$
$flavor? \in \mathrm{dom}\, Thread\_state\_count$

$thread? \neq \underline{a}ctive\_thread(cpu??)$
$(thread?, flavor?) \in \mathrm{dom}\, \underline{t}hread\_state$
$old\_state\_cnt? \geq Thread\_state\_count(flavor?)$

$$NotC1ThreadGetStateGoodArgs \mathrel{\widehat=} ThreadMachineState \wedge ThreadsAndProcessors \wedge \neg\, C1ThreadGetStateGoodArgs$$

9.10.4 Return Values

Tables 30–32 describe the values returned at the completion of the request and the conditions under which each value is returned. The design does not specify the values of $old\_state!$ and $old\_state\_cnt!$ when an error occurs. These values therefore depend on the implementation algorithm and we leave them unspecified.

$return$	C1
$Kern\_success$	T
$Kern\_invalid\_argument$	F

Table 30: Return Values for **thread_get_state**

$old\_state!$	C1
$\underline{t}hread\_state(thread?, flavor?)$	T
—	F

Table 31: Return Values for **thread_get_state**

$old\_state\_cnt!$	C1
$Thread\_state\_count(flavor?)$	T
—	F

Table 32: Return Values for **thread_get_state**

$$
\begin{array}{l}
\textit{RVThreadGetStateGood} \\
\hline
C1ThreadGetStateGoodArgs \\
ThreadMachineState \\
ThreadGetStateOutputs \\
\hline
old\_state\_cnt! = Thread\_state\_count(flavor?) \\
old\_state! = \underline{t}hread\_state(thread?, flavor?) \\
return! = Kern\_success
\end{array}
$$

$$
\begin{array}{l}
\textit{RVThreadGetStateInvalidArgument} \\
\hline
NotC1ThreadGetStateGoodArgs \\
ThreadGetStateOutputs \\
\hline
return! = Kern\_invalid\_argument
\end{array}
$$

9.10.5 State Changes

A successful **thread_get_state** request gets the state of the thread to the supplied state information. The run state of the thread may also change since the request must ensure that the thread is temporarily suspended and then perhaps restart it.

$$
\begin{array}{l}
\textit{ThreadGetStateState} \\
\hline
\Delta\ Threads \\
\Xi\ TasksAndThreads \\
\Xi\ ThreadPri \\
\Xi\ ThreadSchedPolicy \\
\Xi\ ThreadInstruction \\
\Delta\ ThreadExecStatus \\
\Xi\ ThreadStatistics \\
\Xi\ ThreadMachineState \\
\Xi\ Exist \\
\Xi\ SpecialPurposePorts \\
\Xi\ ThreadAndProcessorSet \\
ThreadInvariants \\
ThreadGetStateInputs \\
target\_thread? : THREAD \\
\hline
ThreadDoWaitThenRelease[target\_thread?/stopping\_thread] \\
\underline{s}wapped\_threads' = \underline{s}wapped\_threads \\
\underline{i}dle\_threads' = \underline{i}dle\_threads \\
\underline{t}hread\_suspend\_count' = \underline{t}hread\_suspend\_count \\
\underline{t}hreads\_wired' = \underline{t}hreads\_wired
\end{array}
$$

9.10.6 Complete Request

The following schemas define the general form of a **thread_get_state** request.

$$
\begin{array}{l}
\textit{ProcessingThreadGetState} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_get\_state\_id
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$ThreadGetStateGood \mathrel{\widehat=} (RVThreadGetStateGood \land ThreadGetStateState) \\ \gg ThreadGetStateReply$$

An unsuccessful request returns an error status.

$$ThreadGetStateBad \mathrel{\widehat=} RVThreadGetStateInvalidArgument \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadGetState \mathrel{\widehat=} ThreadGetStateGood \lor ThreadGetStateBad$$

The full specification for kernel processing of a validated **thread_get_state** request consists of processing the request followed by its execution.

$$ThreadGetState \mathrel{\widehat=} ProcessingThreadGetState \fatsemi ExecuteThreadGetState$$

9.11 thread_info

The request **thread_info** returns a specified type of information about a thread. The two valid choices for information types are the thread's execution status and statistics, or its scheduling parameters.

9.11.1 Client Interface

kern_return_t **thread_info**

(mach_port_t	*target_thread_name*,
int	*flavor*,
thread_info_t	*thread_info*,
mach_msg_type_number_t*	*thread_infoCnt*);

9.11.1.1 Input Parameters The following input parameters are provided by the client of a **thread_info** request:

- $target\_thread\_name?$ — the client's name for the thread whose information is to be returned
- $flavor?$ — the type of information that is to be returned. The recognized information types are $Thread\_basic\_info$ and $Thread\_sched\_info$.
- $thread\_infoCnt?$ — the maximum amount of return information that can be handled by the client

```
ThreadInfoClientInputs
target_thread_name? : NAME
flavor? : THREAD_INFO_TYPE
thread_infoCnt? : ℕ
```

A **thread_info** request is invoked by sending a message to the port indicated by *target_thread_name?* that has the operation field set to *Thread_info_id* and has a body consisting of *flavor?* and *thread_infoCnt?*.

```
InvokeThreadInfo
InvokeMachMsg
ThreadInfoClientInputs
─────────
name? = target_thread_name?
operation? = Thread_info_id
msg_body = Thread_info_type_and_count_to_text(flavor?, thread_infoCnt?)
```

9.11.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_info** request:

- *return!* — the status of the request
- *thread_info!* — the information about the target thread
- *thread_infoCnt!* — the size of the information returned in *thread_info!*

```
ThreadInfoClientOutputs
return! : KERNEL_RETURN
thread_info! : THREAD_INFO
thread_infoCnt! : ℕ
```

```
ThreadInfoReceiveReply
InvokeMachMsgRcv
ThreadInfoClientOutputs
─────────
(thread_info!, thread_infoCnt!, return!)
    = Text_to_info_and_count_and_status(msg_body)
```

9.11.2 Kernel Interface

9.11.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_info** request:

- *target_thread?* — the thread whose information is to be returned
- *flavor?* — the type of information that is to be returned. The recognized information types are *Thread_basic_info* and *Thread_sched_info*.

- $thread\_infoCnt?$ — the maximum amount of return information that can be handled by the client

$$\begin{array}{|l}
\multicolumn{1}{l}{__ ThreadInfoInputs __} \\
target_thread? : THREAD \\
flavor? : THREAD_INFO_TYPE \\
thread_infoCnt? : \mathbb{N} \\
\hline
\end{array}$$

9.11.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_info** request:

- $return!$ — the status of the request
- $thread\_info!$ — the information about the target thread
- $thread\_infoCnt!$ — the size of the information returned in $thread\_info!$

$$\begin{array}{|l}
\multicolumn{1}{l}{__ ThreadInfoOutputs __} \\
return! : KERNEL_RETURN \\
thread_info! : THREAD_INFO \\
thread_infoCnt! : \mathbb{N} \\
\hline
\end{array}$$

Upon completion of the processing of a **thread_info** request a reply message is built from the output parameters.

$$\begin{array}{|l}
\multicolumn{1}{l}{__ ThreadInfoReply __} \\
RequestOnlyObserves \\
thread_info? : THREAD_INFO \\
thread_infoCnt? : \mathbb{N} \\
\hline
reply? = Return_thread_info(thread_info?, thread_infoCnt?) \\
\hline
\end{array}$$

The information returned for $Thread\_basic\_info$ is comprised of the following items:

- $user\_time\_value$ — the total user run time for the thread
- $system\_time\_value$ — the total system run time for the thread
- cpu_time_value — the cpu time used for the thread
- $thread\_base\_priority\_value$ — the base user-setable priority for the thread
- $thread\_sched\_priority\_value$ — the priority value used by the system to make scheduling decisions. This is calculated by the system based on the $thread\_base\_priority\_value$, the scheduling policy for the thread, and other system conditions.
- run_state_value — a set which either contains one of the values $Running$, $Stopped$, $Waiting$, $Uninterruptible$, and $Halted$, or is empty
- $flags$ — a set which either contains one of the values $Thread\_flags\_swapped$ or $Thread\_flags\_idle$, or is empty

- $thread\_suspend\_count\_value$ — a thread may execute user level instructions only if this value is zero
- $sleep\_time\_value$ — the amount of time for which a thread has been sleeping

$$THREAD\_FLAGS ::= Thread\_flags\_swapped \mid Thread\_flags\_idle$$

ThreadBasicInfo

$user\_time\_value : \mathbb{N}$
$system\_time\_value : \mathbb{N}$
$cpu\_time\_value : \mathbb{N}$
$thread\_base\_priority\_value : \mathbb{Z}$
$thread\_sched\_priority\_value : \mathbb{Z}$
$run\_state\_value : \mathbb{P}\ RUN\_STATES$
$flags : \mathbb{P}\ THREAD\_FLAGS$
$thread\_suspend\_count\_value : \mathbb{N}$
$sleep\_time\_value : \mathbb{N}$

$\#run\_state\_value \leq 1$
$\#flags \leq 1$

The information returned for $Thread\_sched\_info$ is comprised of the following items:

- $thread\_policy\_value$ — the scheduling policy in force for the thread
- $thread\_sched\_policy\_data\_value$ — policy-specific data that may influence the functioning of the policy in force
- $thread\_base\_priority\_value$ — see above
- $thread\_max\_priority\_value$ — the highest priority to which $thread\_base\_priority\_value$ can be set
- $thread\_sched\_priority\_value$ — see above
- $depressed\_indicator\_value$ — equal to $True$ if the thread's scheduling priority is currently depressed to the lowest possible value, and equal to $False$ otherwise. Priority depression is accomplished via **thread_switch** or **swtch_pri**.
- $priority\_before\_depression\_value$ — if the thread's scheduling priority is currently depressed, the scheduling priority of the thread before it was depressed; otherwise, equal to $thread\_base\_priority\_value$

ThreadSchedInfo

$thread\_policy\_value : SCHED\_POLICY$
$thread\_sched\_policy\_data\_value : SCHED\_POLICY\_DATA$
$thread\_base\_priority\_value : \mathbb{Z}$
$thread\_max\_priority\_value : \mathbb{Z}$
$thread\_sched\_priority\_value : \mathbb{Z}$
$depressed\_indicator\_value : BOOLEAN$
$priority\_before\_depression\_value : \mathbb{Z}$

The actual space required for each type of thread information is represented by the constants $Thread\_basic\_info\_count$ and $Thread\_sched\_info\_count$.

$$
\begin{array}{|l}
Thread\_basic\_info\_count : \mathbb{N} \\
Thread\_sched\_info\_count : \mathbb{N}
\end{array}
$$

9.11.3 Request Criteria

The following criteria are defined for the **thread_info** request.

- **C1 —** Basic information (i.e., execution statistics, status and priorities) is requested, and the client has provided enough space to hold the information.

$$
\begin{array}{|l}
\multicolumn{1}{l}{C1BasicInfo} \\
\hline
flavor? : THREAD\_INFO\_TYPE \\
thread\_infoCnt? : \mathbb{N} \\
\hline
flavor? = Thread\_basic\_info \\
thread\_infoCnt? \geq Thread\_basic\_info\_count \\
\hline
\end{array}
$$

$$NotC1BasicInfo \mathrel{\widehat=} \neg\ C1BasicInfo$$

- **C2 —** Scheduling information (i.e., priorities and policies) is requested, and the client has provided enough space to hold the information.

$$
\begin{array}{|l}
\multicolumn{1}{l}{C2SchedInfo} \\
\hline
flavor? : THREAD\_INFO\_TYPE \\
thread\_infoCnt? : \mathbb{N} \\
\hline
flavor? = Thread\_sched\_info \\
thread\_infoCnt? \geq Thread\_sched\_info\_count \\
\hline
\end{array}
$$

$$NotC2SchedInfo \mathrel{\widehat=} \neg\ C2SchedInfo$$

9.11.4 Return Values

Tables 33–35 describe the values returned at the completion of the request and the conditions under which each value is returned. The specification does not state what should be returned in $thread\_infoCnt!$ and $thread\_info!$ when there is an error. These values depend in the implementation, and we leave them unspecified. Note that C1 and C2 cannot simultaneously be true.

$thread\_infoCnt!$	C1	C2
$Thread\_basic\_info\_count$	T	F
$Thread\_sched\_info\_count$	F	T
—	F	F

Table 33: Return Values for **thread_info**

$thread\_info!$	C1	C2
$Format\_basic\_info(basic\_info)$	T	F
$Format\_sched\_info(sched\_info)$	F	T
—	F	F

Table 34: Return Values for **thread_info**

$return!$	C1	C2
$Kern\_success$	T	F
$Kern\_success$	F	T
$Kern\_invalid\_argument$	F	F

Table 35: Return Values for **thread_info**

Each of the two types of information must be reformatted into $thread\_info!$ to be returned. This formatting is represented by the functions $Format\_basic\_info$ and $Format\_sched\_info$.

$$Format\_basic\_info : ThreadBasicInfo \longrightarrow THREAD\_INFO$$
$$Format\_sched\_info : ThreadSchedInfo \longrightarrow THREAD\_INFO$$

The kernel maintains for each thread a set of run states drawn from the values $Halted$, $Running$, $Uninterruptible$, $Stopped$ and $Waiting$. At most one of these values is returned with the basic information. The value selected is the first one in the above list that is in the run state of the thread. In the event the run state set is empty, an empty set is returned. The function $Primary\_run\_state$ represents this mapping from internal run state to basic information.

Similarly, the kernel will only return one of the flags $Thread\_flags\_swapped$ or $Thread\_flags\_idle$. If $thread?$ is in $\underline{s}wapped\_threads$, then $Thread\_flags\_swapped$ is always returned regardless of the value of $\underline{i}dle\_threads$.

$$Primary\_run\_state : \mathbb{P}\ RUN\_STATES \longrightarrow \mathbb{P}\ RUN\_STATES$$
$$Run\_state\_order : \mathrm{seq}\ RUN\_STATES$$

$$Run\_state\_order = \langle Halted, Running, Uninterruptible, Stopped, Waiting \rangle$$
$$\forall\, r : \mathbb{P}\ RUN\_STATES$$
$$\bullet\ Primary\_run\_state(r) = (Run\_state\_order \upharpoonright r)(\!|\{1\}|\!)$$

Review Note:
It is important to take the relational image under $\{1\}$ rather than taking the head of the sequence since the sequence might be empty.

$$
\begin{array}{l}
\textbf{RVThreadBasicInfo} \\
\hline
ThreadStatistics \\
ThreadPri \\
ThreadExecStatus \\
C1BasicInfo \\
NotC2SchedInfo \\
ThreadInfoOutputs \\
thread? : THREAD \\
\hline
(\exists\, basic\_info : ThreadBasicInfo \\
\bullet\ basic\_info.user\_time\_value = \underline{u}ser\_time(thread?) \\
\land\ basic\_info.system\_time\_value = \underline{s}ystem\_time(thread?) \\
\land\ basic\_info.cpu\_time\_value = \underline{c}pu\_time(thread?) \\
\land\ basic\_info.thread\_base\_priority\_value = \underline{t}hread\_priority(thread?) \\
\land\ basic\_info.thread\_sched\_priority\_value = \underline{t}hread\_sched\_priority(thread?) \\
\land\ basic\_info.run\_state\_value = Primary\_run\_state(\underline{r}un\_state(thread?)) \\
\land\ basic\_info.flags = \mathbf{if}\ thread? \in \underline{s}wapped\_threads \\
\qquad \mathbf{then}\ \{Thread\_flags\_swapped\} \\
\qquad \mathbf{else}\ (\mathbf{if}\ thread? \in \underline{i}dle\_threads \\
\qquad\qquad \mathbf{then}\ \{Thread\_flags\_idle\} \\
\qquad\qquad \mathbf{else}\ \emptyset) \\
\land\ basic\_info.thread\_suspend\_count\_value = \underline{t}hread\_suspend\_count(thread?) \\
\land\ basic\_info.sleep\_time\_value = \underline{s}leep\_time(thread?) \\
\land\ thread\_info! = Format\_basic\_info(basic\_info)) \\
\\
thread\_infoCnt! = Thread\_basic\_info\_count \\
return! = Kern\_success \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{RVThreadSchedInfo} \\
\hline
ThreadPri \\
ThreadSchedPolicy \\
NotC1BasicInfo \\
C2SchedInfo \\
ThreadInfoOutputs \\
thread? : THREAD \\
\hline
(\exists\, sched\_info : ThreadSchedInfo \\
\bullet\ sched\_info.thread\_policy\_value = \underline{t}hread\_sched\_policy(thread?) \\
\land\ sched\_info.thread\_sched\_policy\_data\_value \\
\qquad = \underline{t}hread\_sched\_policy\_data(thread?) \\
\land\ sched\_info.thread\_base\_priority\_value = \underline{t}hread\_priority(thread?) \\
\land\ sched\_info.thread\_max\_priority\_value = \underline{t}hread\_max\_priority(thread?) \\
\land\ sched\_info.thread\_sched\_priority\_value = \underline{t}hread\_sched\_priority(thread?) \\
\land\ sched\_info.depressed\_indicator\_value = \mathbf{if}\ thread? \in \underline{d}epressed\_threads \\
\qquad \mathbf{then}\ True \\
\qquad \mathbf{else}\ False \\
\land\ sched\_info.priority\_before\_depression\_value = \underline{p}riority\_before\_depression(thread?) \\
\land\ thread\_info! = Format\_sched\_info(sched\_info)) \\
\\
thread\_infoCnt! = Thread\_sched\_info\_count \\
return! = Kern\_success \\
\hline
\end{array}
$$

$$\begin{array}{|l} \hline \multicolumn{1}{l}{\textit{RVThreadInfoBad}} \\ NotC1BasicInfo \\ NotC2SchedInfo \\ ThreadInfoOutputs \\ \hline return! = Kern\_invalid\_argument \\ \hline \end{array}$$

9.11.5 State Changes

A **thread_info** request does not make any state changes since it only observes the system state.

9.11.6 Complete Request

The following schemas define the general form of a **thread_info** request.

$$\begin{array}{|l} \hline \multicolumn{1}{l}{\textit{ProcessingThreadInfo}} \\ ProcessThreadViaThreadPortRequestGood \\ \hline operation? = Thread\_info\_id \\ \hline \end{array}$$

A successful request creates a kernel reply.

$$ThreadInfoGood \mathrel{\widehat{=}} (RVThreadBasicInfo \lor RVThreadSchedInfo) \\ \quad \gg ThreadInfoReply$$

An unsuccessful request returns an error status.

$$ThreadInfoBad \mathrel{\widehat{=}} RVThreadInfoBad \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadInfo \mathrel{\widehat{=}} ThreadInfoGood \lor ThreadInfoBad$$

The full specification for kernel processing of a validated **thread_info** request consists of processing the request followed by its execution.

$$ThreadInfo \mathrel{\widehat{=}} ProcessingThreadInfo \mathbin{⨟} ExecuteThreadInfo$$

9.12 thread_max_priority

The request **thread_max_priority** sets the maximum scheduling priority of a specified thread. This value limits the value to which the user can set the priority of the thread (using **thread_priority**). Since the client thread must have access to the control port of the processor set to which the thread is assigned, the request may set the maximum priority to any legal value including one that is higher than the current value. This contrasts with **thread_priority** which does not require control port access and can only lower the maximum priority.

9.12.1 Client Interface

```
kern_return_t thread_max_priority
              (mach_port_t      thread_name,
               mach_port_t      processor_set_name,
               int              priority);
```

9.12.1.1 Input Parameters The following input parameters are provided by the client of a **thread_max_priority** request:

- $thread\_name?$ — the client's name for the the thread whose maximum priority is to be set
- $processor\_set\_name?$ — the client's name for the control port of the processor set to which the target thread is currently assigned
- $priority?$ — the desired maximum priority

$$
\begin{array}{|l}
\hline
\textit{ThreadMaxPriorityClientInputs} \\
thread\_name? : NAME \\
processor\_set\_name? : NAME \\
priority? : \mathbb{Z} \\
\hline
\end{array}
$$

A **thread_max_priority** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_max\_priority\_id$ and has a body consisting of $processor\_set\_name?$ and $priority?$.

$$
\begin{array}{|l}
\hline
\textit{InvokeThreadMaxPriority} \\
InvokeMachMsg \\
ThreadMaxPriorityClientInputs \\
\hline
name? = thread\_name? \\
operation? = Thread\_max\_priority\_id \\
msg\_body = Name\_and\_number\_to\_text(processor\_set\_name?, priority?) \\
\hline
\end{array}
$$

9.12.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_max_priority** request:

- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{ThreadMaxPriorityClientOutputs} \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ThreadMaxPriorityReceiveReply} \\
InvokeMachMsgRcv \\
ThreadMaxPriorityClientOutputs \\
\hline
return! = Text\_to\_status(msg\_body) \\
\hline
\end{array}
$$

9.12.2 Kernel Interface

9.12.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_max_priority** request:

- $thread?$ — the thread whose maximum priority is to be set
- $processor\_set?$ — the processor set to which the target thread is currently assigned
- $priority?$ — the desired maximum priority

$$
\begin{array}{l}
\textit{ThreadMaxPriorityInputs} \\
\hline
thread? : THREAD \\
processor\_set? : PROCESSOR\_SET \\
priority? : \mathbb{Z} \\
\hline
\end{array}
$$

9.12.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_max_priority** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadMaxPriorityOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

9.12.3 Request Criteria

The following criteria are defined for the **thread_max_priority** request.

Review Note:
It is assumed here that the existence of $processor\_set?$ has been verified by the IPC processing of the request. However, since an arbitrary time delay might occur between the IPC processing and the kernel request processing, we should probably have an additional check here that the $processor\_set?$ still exists.

- **C1 —** The parameter $priority?$ is a valid priority level.

$$
\begin{array}{l}
\textit{C1ThreadMaxPriorityValidArguments} \\
\hline
priority? : \mathbb{Z} \\
\hline
priority? \in Priority\_levels \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1ThreadMaxPriorityValidArguments \\
\quad \widehat{=} \neg\ C1ThreadMaxPriorityValidArguments
\end{array}
$$

- **C2 —** The parameter $processor\_set?$ is the processor set to which $thread?$ is assigned.

$C2ThreadMaxPriorityAssignedProcessorSet$
$ThreadAndProcessorSet$
$thread? : THREAD$
$processor\_set? : PROCESSOR\_SET$

$(thread?, processor\_set?) \in thread\_assigned\_to$

$$NotC2ThreadMaxPriorityAssignedProcessorSet \mathrel{\widehat=} ThreadAndProcessorSet \land \neg\, C2ThreadMaxPriorityAssignedProcessorSet$$

9.12.4 Return Values

Table 36 describes the values returned at the completion of the request and the conditions under which each value is returned. In the case where both C1 and C2 are false, we assume $Kern\_invalid\_argument$ is returned.

Review Note:
The prototype checks the conditions in the order C1, C2.

$return!$	C1	C2
$Kern\_success$	T	T
$Kern\_failure$	T	F
$Kern\_invalid\_argument$	F	-

Table 36: Return Values for **thread_max_priority**

$RVThreadMaxPriorityGood$
$C1ThreadMaxPriorityValidArguments$
$C2ThreadMaxPriorityAssignedProcessorSet$
$ThreadMaxPriorityOutputs$

$return! = Kern\_success$

$RVThreadMaxPriorityWrongProcessorSet$
$C1ThreadMaxPriorityValidArguments$
$NotC2ThreadMaxPriorityAssignedProcessorSet$
$ThreadMaxPriorityOutputs$

$return! = Kern\_failure$

$RVThreadMaxPriorityInvalidArgument$
$NotC1ThreadMaxPriorityValidArguments$
$ThreadMaxPriorityOutputs$

$return! = Kern\_invalid\_argument$

9.12.5 State Changes

A successful **thread_max_priority** request sets the maximum priority of the thread as requested. If the thread's priority is higher than this new maximum value, it is reset to the new maximum value. If the thread is currently depressed and if its priority before depression is higher than the new maximum, then the priority before depression is reset to the new maximum value. In this way the modification of $\underline{t}hread\_max\_priority$ will be reflected in the priority of the thread when its priority depression is removed. Note that if the thread is currently depressed $\underline{t}hread\_priority(thread?)$ will be equal to the lowest possible priority. Thus, $Lowest\_priority\{\underline{t}hread\_priority(thread?), priority?\}\} = \underline{t}hread\_priority(thread?)$ and no change is made to $\underline{t}hread\_priority$. The thread's current scheduling priority may or may not change as a result of this request, so we state no constraint on the value of $\underline{t}hread\_sched\_priority$.

$$
\begin{array}{l}
\textit{ThreadMaxPriorityState} \\
\hline
\Delta\ Threads \\
\Xi\ TasksAndThreads \\
\Delta\ ThreadPri \\
\Xi\ ThreadSchedPolicy \\
\Xi\ ThreadInstruction \\
\Xi\ ThreadExecStatus \\
\Xi\ ThreadStatistics \\
\Xi\ ThreadMachineState \\
\Xi\ Exist \\
\Xi\ SpecialPurposePorts \\
\Xi\ ThreadAndProcessorSet \\
ThreadInvariants \\
thread? : THREAD \\
priority? : \mathbb{Z} \\
\hline
\underline{d}epressed\_threads' = \underline{d}epressed\_threads \\
\underline{t}hread\_max\_priority' = \underline{t}hread\_max\_priority \oplus \{thread? \mapsto priority?\} \\
\underline{p}riority\_before\_depression' = \mathbf{if}\ thread? \in \underline{d}epressed\_threads \\
\quad \mathbf{then}\ \underline{p}riority\_before\_depression \oplus \{thread? \\
\qquad \mapsto Lowest\_priority\{\underline{p}riority\_before\_depression(thread?), priority?\}\} \\
\quad \mathbf{else}\ \underline{p}riority\_before\_depression \oplus \{thread? \mapsto \underline{t}hread\_priority'(thread?)\} \\
\underline{t}hread\_priority' = \underline{t}hread\_priority \\
\quad \oplus \{thread? \mapsto Lowest\_priority\{\underline{t}hread\_priority(thread?), priority?\}\} \\
\hline
\end{array}
$$

9.12.6 Complete Request

The following schemas define the general form of a **thread_max_priority** request.

$$
\begin{array}{l}
\textit{ProcessingThreadMaxPriority} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_max\_priority\_id \\
\hline
\end{array}
$$

A successful request makes the state changes described in the previous section.

$$
\begin{array}{l}
ThreadMaxPriorityGood \mathrel{\widehat=} (RVThreadMaxPriorityGood \land ThreadMaxPriorityState) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

An unsuccessful request returns an error status.

$$ThreadMaxPriorityBad \mathrel{\widehat=} (RVThreadMaxPriorityWrongProcessorSet \lor RVThreadMaxPriorityInvalidArgument) \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadMaxPriority \mathrel{\widehat=} ThreadMaxPriorityGood \lor ThreadMaxPriorityBad$$

The full specification for kernel processing of a validated **thread_max_priority** request consists of processing the request followed by its execution.

$$ThreadMaxPriority \mathrel{\widehat=} ProcessingThreadMaxPriority \fatsemi ExecuteThreadMaxPriority$$

9.13 thread_policy

The request **thread_policy** sets the scheduling policy for a specified thread. This value is used by the system (together with the thread priority and current conditions) to determine the current scheduling priority of the thread.

9.13.1 Client Interface

```
kern_return_t thread_policy
        (mach_port_t        thread_name,
         int                policy,
         int                data);
```

9.13.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_policy** request:

- $thread\_name?$ — the client's name for the thread whose scheduling policy is to be set
- $policy?$ — the desired scheduling policy
- $data?$ — policy specific data which may influence the operation of the scheduling policy

$$\begin{array}{l} \text{ThreadPolicyClientInputs} \\ \hline thread\_name? : NAME \\ policy? : SCHED\_POLICY \\ data? : SCHED\_POLICY\_DATA \\ \hline \end{array}$$

A **thread_policy** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_policy\_id$ and has a body consisting of $policy?$ and $data?$.

$$\begin{array}{l}
\textit{InvokeThreadPolicy} \\
\hline
\textit{InvokeMachMsg} \\
\textit{ThreadPolicyClientInputs} \\
\hline
name? = thread_name? \\
operation? = Thread_policy_id \\
msg_body = Policy_and_data_to_text(policy?, data?)
\end{array}$$

9.13.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_policy** request:

- $return!$ — the status of the request

$$\begin{array}{l}
\textit{ThreadPolicyClientOutputs} \\
\hline
return! : KERNEL_RETURN
\end{array}$$

$$\begin{array}{l}
\textit{ThreadPolicyReceiveReply} \\
\hline
\textit{InvokeMachMsgRcv} \\
\textit{ThreadPolicyClientOutputs} \\
\hline
return! = Text_to_status(msg_body)
\end{array}$$

9.13.2 Kernel Interface

9.13.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_policy** request:

- $thread?$ — the thread whose scheduling policy is to be set
- $policy?$ — the desired scheduling policy
- $data?$ — policy specific data which may influence the operation of the scheduling policy

$$\begin{array}{l}
\textit{ThreadPolicyInputs} \\
\hline
thread? : THREAD \\
policy? : SCHED_POLICY \\
data? : SCHED_POLICY_DATA
\end{array}$$

9.13.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_policy** request:

- $return!$ — the status of the request

$$\begin{array}{l}
\textit{ThreadPolicyOutputs} \\
\hline
return! : KERNEL_RETURN
\end{array}$$

9.13.3 Request Criteria

The following criteria are defined for the **thread_policy** request.

- **C1 —** The parameter $policy?$ is an existing policy.

$$\begin{array}{l} \textit{C1ThreadPolicyValidPolicy} \\ \hline ThreadSchedPolicy \\ policy? : SCHED\_POLICY \\ \hline policy? \in \underline{s}upported\_sp \end{array}$$

$$NotC1ThreadPolicyValidPolicy \mathrel{\widehat=} ThreadSchedPolicy \land \neg\ C1ThreadPolicyValidPolicy$$

- **C2 —** The parameter $policy?$ is a permitted scheduling policy for the processor set to which $thread?$ is assigned.

$$\begin{array}{l} \textit{C2ThreadPolicyProcessorSetPermits} \\ \hline ThreadAndProcessorSet \\ thread? : THREAD \\ policy? : SCHED\_POLICY \\ \hline thread? \in \operatorname{dom} thread\_assigned\_to \\ thread\_assigned\_to(thread?) \in \operatorname{dom} \underline{e}nabled\_sp \\ \\ policy? \in \underline{e}nabled\_sp(thread\_assigned\_to(thread?)) \end{array}$$

$$NotC2ThreadPolicyProcessorSetPermits \mathrel{\widehat=} ThreadAndProcessorSet \\ \quad \land \neg\ C2ThreadPolicyProcessorSetPermits$$

9.13.4 Return Values

Table 37 describes the values returned at the completion of the request and the conditions under which each value is returned. In the case where both C1 and C2 are false, we assume $Kern\_invalid\_argument$ is returned.

Review Note:
The prototype checks the conditions in the order C1, C2.

$return!$	C1	C2
$Kern\_success$	T	T
$Kern\_failure$	T	F
$Kern\_invalid\_argument$	F	-

Table 37: Return Values for **thread_policy**

$$\begin{array}{l}
\textit{RVThreadPolicyGood} \\
\hline
C1\,ThreadPolicyValidPolicy \\
C2\,ThreadPolicyProcessorSetPermits \\
ThreadPolicyOutputs \\
\hline
return! = Kern_success
\end{array}$$

$$\begin{array}{l}
\textit{RVThreadPolicyNotPermitted} \\
\hline
C1\,ThreadPolicyValidPolicy \\
NotC2\,ThreadPolicyProcessorSetPermits \\
ThreadPolicyOutputs \\
\hline
return! = Kern_failure
\end{array}$$

$$\begin{array}{l}
\textit{RVThreadPolicyInvalidPolicy} \\
\hline
NotC1\,ThreadPolicyValidPolicy \\
ThreadPolicyOutputs \\
\hline
return! = Kern_invalid_argument
\end{array}$$

9.13.5 State Changes

A successful **thread_policy** request sets the target thread's policy and policy specific data as requested.

$$\begin{array}{l}
\textit{ThreadPolicyNewPolicy} \\
\hline
\Delta\ ThreadSchedPolicy \\
thread? : THREAD \\
policy? : SCHED_POLICY \\
data? : SCHED_POLICY_DATA \\
\hline
\underline{t}hread_sched_policy' = \underline{t}hread_sched_policy \oplus \{thread? \mapsto policy?\} \\
\underline{t}hread_sched_policy_data' = \underline{t}hread_sched_policy_data \\
\quad \oplus \{\ thread? \mapsto data?\ \} \\
\underline{s}upported_sp' = \underline{s}upported_sp
\end{array}$$

The priority and maximum priority of the thread are not modified. However, the thread's current scheduling priority may or may not change as a result of this request, so we state no constraint on the value of $\underline{t}hread\_sched\_priority$.

$$\begin{array}{l}
\textit{ThreadPolicyPriority} \\
\hline
\Delta\ ThreadPri \\
\hline
\underline{t}hread_priority' = \underline{t}hread_priority \\
\underline{t}hread_max_priority' = \underline{t}hread_max_priority \\
\underline{d}epressed_threads' = \underline{d}epressed_threads \\
\underline{p}riority_before_depression' = \underline{p}riority_before_depression
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{ThreadPolicyState} \\
\hline
\textit{ThreadPolicyNewPolicy} \\
\textit{ThreadPolicyPriority} \\
\Delta\ \textit{Threads} \\
\Xi\ \textit{TasksAndThreads} \\
\Xi\ \textit{ThreadInstruction} \\
\Xi\ \textit{ThreadExecStatus} \\
\Xi\ \textit{ThreadStatistics} \\
\Xi\ \textit{ThreadMachineState} \\
\Xi\ \textit{Exist} \\
\Xi\ \textit{SpecialPurposePorts} \\
\Xi\ \textit{ThreadAndProcessorSet} \\
\textit{ThreadInvariants} \\
\hline
\end{array}$$

9.13.6 Complete Request

The following schema defines the general form of **thread_policy**.

$$\begin{array}{|l}
\hline
\textit{ProcessingThreadPolicy} \\
\hline
\textit{ProcessThreadViaThreadPortRequestGood} \\
\hline
\textit{operation?} = \textit{Thread_policy_id} \\
\hline
\end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$\begin{array}{l}
\textit{ThreadPolicyGood} \mathrel{\widehat=} (\textit{RVThreadPolicyGood} \land \textit{ThreadPolicyState}) \\
\quad \gg \textit{RequestReturnOnlyStatus}
\end{array}$$

An unsuccessful request returns an error status.

$$\begin{array}{l}
\textit{ThreadPolicyBad} \\
\quad \mathrel{\widehat=} (\textit{RVThreadPolicyNotPermitted} \lor \textit{RVThreadPolicyInvalidPolicy}) \\
\quad \gg \textit{RequestNoOp}
\end{array}$$

Execution of the request consists of a good execution or an error execution.

$$\begin{array}{l}
\textit{ExecuteThreadPolicy} \\
\quad \mathrel{\widehat=} \textit{ThreadPolicyGood} \lor \textit{ThreadPolicyBad}
\end{array}$$

The full specification for kernel processing of a validated **thread_policy** request consists of processing the request followed by its execution.

$$\textit{ThreadPolicy} \mathrel{\widehat=} \textit{ProcessingThreadPolicy} \mathbin{;} \textit{ExecuteThreadPolicy}$$

9.14 thread_priority

The request **thread_priority** sets the priority of a specified thread. This priority is used by the system (together with the thread scheduling policy and current conditions) to determine the current scheduling priority of the thread used when scheduling threads to run. If the priority of the thread is currently depressed, the priority before depression will be reset instead so that this request will take effect when the priority depression is aborted. Optionally, the request may also *lower* the thread maximum priority, which limits the value of the priority. The request **thread_max_priority** also changes the priority and maximum priority. However, **thread_max_priority** requires access to the control port of the processor set to which the thread is assigned, and it may *raise* the maximum priority.

9.14.1 Client Interface

```
kern_return_t thread_priority
        (mach_port_t        thread_name,
         int                priority,
         boolean_t          set_max);
```

9.14.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_priority** request:

- $thread\_name?$ — the client's name for the thread whose priority is to be set
- $priority?$ — the desired priority
- $set\_max?$ — a boolean parameter equal to $True$ if the thread's maximum priority value should also be set (lowered) to $priority?$

ThreadPriorityClientInputs

$thread\_name? : NAME$
$priority? : \mathbb{Z}$
$set\_max? : BOOLEAN$

A **thread_priority** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_priority\_id$ and has a body consisting of $priority?$ and $set\_max?$.

InvokeThreadPriority

$InvokeMachMsg$
$ThreadPriorityClientInputs$

$name? = thread\_name?$
$operation? = Thread\_priority\_id$
$msg\_body = Number\_and\_boolean\_to\_text(priority?, set\_max?)$

9.14.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_priority** request:

- $return!$ — the status of the request

$$\text{ThreadPriorityClientOutputs}$$
$$return! : KERNEL\_RETURN$$

$$\text{ThreadPriorityReceiveReply}$$
$$InvokeMachMsgRcv$$
$$ThreadPriorityClientOutputs$$
$$\rule{2cm}{0.4pt}$$
$$return! = Text\_to\_status(msg\_body)$$

9.14.2 Kernel Interface

9.14.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_priority** request:

- $thread?$ — the thread whose priority is to be set
- $priority?$ — the desired priority
- $set\_max?$ — a boolean parameter equal to $True$ if the thread's maximum priority value should also be set to $priority?$

$$\text{ThreadPriorityInputs}$$
$$thread? : THREAD$$
$$priority? : \mathbb{Z}$$
$$set\_max? : BOOLEAN$$

9.14.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_priority** request:

- $return!$ — the status of the request

$$\text{ThreadPriorityOutputs}$$
$$return! : KERNEL\_RETURN$$

9.14.3 Request Criteria

The following criteria are defined for the **thread_priority** request.

- **C1 —** The parameter $priority?$ is a valid priority level.

$$\text{C1ThreadPriorityValidPriority}$$
$$priority? : \mathbb{Z}$$
$$\rule{2cm}{0.4pt}$$
$$priority? \in Priority\_levels$$

$$NotC1ThreadPriorityValidPriority$$
$$\widehat{=} \neg\ C1ThreadPriorityValidPriority$$

- **C2 —** The new priority is no higher than the maximum priority for $thread?$.

$$\begin{array}{|l} \hline \textit{C2ThreadPriorityAllowedPriority} \\ ThreadPri \\ thread? : THREAD \\ priority? : \mathbb{Z} \\ \hline (priority?, \underline{t}hread\_max\_priority(thread?)) \notin Higher\_priority \\ \hline \end{array}$$

$$NotC2ThreadPriorityAllowedPriority \\ \quad \widehat{=} \ ThreadPri \land \neg \ C2ThreadPriorityAllowedPriority$$

9.14.4 Return Values

Table 38 describes the values returned at the completion of the request and the conditions under which each value is returned. In the case where both C1 and C2 are false we assume $Kern\_invalid\_argument$ is returned.

Review Note:
The prototype checks the conditions in the order C1, C2.

$return!$	C1	C2
$Kern\_success$	T	T
$Kern\_failure$	T	F
$Kern\_invalid\_argument$	F	-

Table 38: Return Values for **thread_priority**

$$\begin{array}{|l} \hline RVThreadPriorityGood \\ C1ThreadPriorityValidPriority \\ C2ThreadPriorityAllowedPriority \\ ThreadPriorityOutputs \\ \hline return! = Kern\_success \\ \hline \end{array}$$

$$\begin{array}{|l} \hline RVThreadPriorityPriorityTooHigh \\ C1ThreadPriorityValidPriority \\ NotC2ThreadPriorityAllowedPriority \\ ThreadPriorityOutputs \\ \hline return! = Kern\_failure \\ \hline \end{array}$$

$$\begin{array}{|l} \hline RVThreadPriorityInvalidPriority \\ NotC1ThreadPriorityValidPriority \\ ThreadPriorityOutputs \\ \hline return! = Kern\_invalid\_argument \\ \hline \end{array}$$

9.14.5 State Changes

A successful **thread_priority** request makes the following changes to the system state. The $\underline{p}riority\_before\_depression$ for the thread is reset to $priority?$. If the thread priority is not currently depressed, $\underline{t}hread\_priority$ for the thread is also reset to $priority?$. Otherwise, $\underline{t}hread\_priority$ does not change. In addition, if $set\_max?$ is $True$, the $\underline{t}hread\_max\_priority$ for the thread will also be reset to $priority?$. Note that the thread's current scheduling priority may or may not change as a result of this request, so we state no constraint on the value of $\underline{t}hread\_sched\_priority$.

$ThreadPriorityInvariants$

$ThreadInvariants$
$\Xi\ TasksAndThreads$
$\Xi\ ThreadSchedPolicy$
$\Xi\ ThreadInstruction$
$\Xi\ ThreadExecStatus$
$\Xi\ ThreadStatistics$
$\Xi\ ThreadMachineState$
$\Xi\ Exist$
$\Xi\ SpecialPurposePorts$
$\Xi\ ThreadAndProcessorSet$

$ThreadPriorityState$

$\Delta\ Threads$
$\Delta\ ThreadPri$
$ThreadPriorityInvariants$
$thread? : THREAD$
$priority? : \mathbb{Z}$
$set\_max? : BOOLEAN$

$\underline{d}epressed\_threads' = \underline{d}epressed\_threads$
$\underline{t}hread\_priority' = \mathbf{if}\ thread? \in \underline{d}epressed\_threads$
$\quad \mathbf{then}\ \underline{t}hread\_priority$
$\quad \mathbf{else}\ \underline{t}hread\_priority \oplus \{thread? \mapsto priority?\}$
$\underline{p}riority\_before\_depression' = \underline{p}riority\_before\_depression$
$\quad \oplus \{thread? \mapsto priority?\}$
$\underline{t}hread\_max\_priority' = \mathbf{if}\ set\_max? = True$
$\quad \mathbf{then}\ \underline{t}hread\_max\_priority \oplus \{thread? \mapsto priority?\}$
$\quad \mathbf{else}\ \underline{t}hread\_max\_priority$

9.14.6 Complete Request

The following schemas define the general form of a **thread_priority** request.

$ProcessingThreadPriority$

$ProcessThreadViaThreadPortRequestGood$

$operation? = Thread\_priority\_id$

A successful request makes the state changes described in the previous section, and creates a kernel reply.

$$ThreadPriorityGood \mathrel{\widehat=} (RVThreadPriorityGood \wedge ThreadPriorityState)$$
$$\gg RequestReturnOnlyStatus$$

An unsuccessful request returns an error status.

$$ThreadPriorityBad$$
$$\mathrel{\widehat=} (RVThreadPriorityPriorityTooHigh \vee RVThreadPriorityInvalidPriority)$$
$$\gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteThreadPriority$$
$$\mathrel{\widehat=} ThreadPriorityGood \vee ThreadPriorityBad$$

The full specification for kernel processing of a validated **thread_priority** request consists of processing the request followed by its execution.

$$ThreadPriority \mathrel{\widehat=} ProcessingThreadPriority \mathbin{;} ExecuteThreadPriority$$

9.15 **thread_resume** and **thread_resume_secure**

The requests **thread_resume** and **thread_resume_secure** decrement the suspend count of a thread by 1. They may impact the thread's run states as a result. The **thread_resume_secure** request (which is used in the secure initiation of threads within a task) expects the parent task to have task creation state $Tcs\_thread\_state\_set$ (see Section 5.7). It modifies the state to Tcs_task_ready.

9.15.1 Client Interface

```
kern_return_t thread_resume
              (mach_port_t                target_thread_name);

kern_return_t thread_resume_secure
              (mach_port_t                target_thread_name);
```

9.15.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_resume** or **thread_resume_secure** request:

- $target\_thread\_name?$ — the client's name for the thread that is to be resumed

$ThreadResumeClientInputs$
$target\_thread\_name? : NAME$

A **thread_resume** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_resume\_id$ and has no body.

$$\begin{array}{|l}
\hline
\textit{InvokeThreadResume} \\
\textit{InvokeMachMsg} \\
\textit{ThreadResumeClientInputs} \\
\hline
name? = target_thread_name? \\
operation? = Thread_resume_id \\
\hline
\end{array}$$

A **thread_resume_secure** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_resume\_secure\_id$ and has no body.

$$\begin{array}{|l}
\hline
\textit{InvokeThreadResumeSecure} \\
\textit{InvokeMachMsg} \\
\textit{ThreadResumeClientInputs} \\
\hline
name? = target_thread_name? \\
operation? = Thread_resume_secure_id \\
\hline
\end{array}$$

9.15.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **thread_resume** or **thread_resume_secure** request:

- $return!$ — the status of the request

$$\begin{array}{|l}
\hline
\textit{ThreadResumeClientOutputs} \\
return! : KERNEL_RETURN \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{ThreadResumeReceiveReply} \\
\textit{InvokeMachMsgRcv} \\
\textit{ThreadResumeClientOutputs} \\
\hline
return! = Text_to_status(msg_body) \\
\hline
\end{array}$$

9.15.2 Kernel Interface

9.15.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_resume** or **thread_resume_secure** request:

- $target\_thread?$ — the thread that is to be resumed

$$\begin{array}{|l}
\hline
\textit{ThreadResumeInputs} \\
target_thread? : THREAD \\
\hline
\end{array}$$

9.15.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_resume** or **thread_resume_secure** request:

- $return!$ — the status of the request

$$\begin{array}{|l}
\hline
\textit{ThreadResumeOutputs} \\
return! : KERNEL_RETURN \\
\hline
\end{array}$$

9.15.3 Request Criteria

The following criteria are defined for the **thread_resume** and **thread_resume_secure** requests.

- **C1 —** The suspend count of the target thread is positive.

$$
\begin{array}{l}
\textit{C1ThreadResumeSuspendCountPositive} \\
\hline
\textit{ThreadExecStatus} \\
\textit{target\_thread?} : \textit{THREAD} \\
\hline
\textit{target\_thread?} \in \operatorname{dom} \underline{t}\textit{hread\_suspend\_count} \\
\\
\underline{t}\textit{hread\_suspend\_count}(\textit{target\_thread?}) > 0
\end{array}
$$

$$
\begin{array}{l}
\textit{NotC1ThreadResumeSuspendCountPositive} \mathrel{\widehat{=}} \\
\quad \textit{ThreadExecStatus} \land \lnot\ \textit{C1ThreadResumeSuspendCountPositive}
\end{array}
$$

- **C2 —** The task creation state of the target thread's owning task must be $\textit{Tcs\_thread\_state\_set}$. This criterion applies only to the **thread_resume_secure** request.

$$
\begin{array}{l}
\textit{C2ThreadResumeThreadStateSet} \\
\hline
\textit{TasksAndThreads} \\
\textit{TaskCreationState} \\
\textit{target\_thread?} : \textit{THREAD} \\
\hline
\textit{target\_thread?} \in \operatorname{dom} \textit{owning\_task} \\
\textit{owning\_task}(\textit{target\_thread?}) \in \operatorname{dom} \underline{t}\textit{ask\_creation\_state} \\
\\
\underline{t}\textit{ask\_creation\_state}(\textit{owning\_task}(\textit{target\_thread?})) = \textit{Tcs\_thread\_state\_set}
\end{array}
$$

$$
\begin{array}{l}
\textit{NotC2ThreadResumeThreadStateSet} \\
\quad \mathrel{\widehat{=}} \textit{TasksAndThreads} \land \textit{TaskCreationState} \land \lnot\ \textit{C2ThreadResumeThreadStateSet}
\end{array}
$$

9.15.4 Return Values

Table 39 describes the values returned at the completion of the **thread_resume** request and the conditions under which each value is returned.

$\textit{return!}$	C1
$\textit{Kern\_success}$	T
$\textit{Kern\_failure}$	F

Table 39: Return Values for **thread_resume**

$$
\begin{array}{l}
\textit{RVThreadResumeGood} \\
\hline
\textit{C1ThreadResumeSuspendCountPositive} \\
\textit{ThreadResumeOutputs} \\
\hline
\textit{return!} = \textit{Kern\_success}
\end{array}
$$

$RVThreadResumeFailure$
$NotC1\,ThreadResumeSuspendCountPositive$
$ThreadResume\,Outputs$

$return! = Kern\_failure$

Table 40 describes the values returned at the completion of the **thread_resume_secure** request and the conditions under which each value is returned. In the case where C1 and C2 are both false we assume $Kern\_insufficient\_permission$ is returned.

Review Note:
C2 is checked first in the prototype.

$return!$	C1	C2
$Kern\_success$	T	T
$Kern\_failure$	F	T
$Kern\_insufficient\_permission$	-	F

Table 40: Return Values for **thread_resume_secure**

$RVThreadResumeSecureGood$
$C1\,ThreadResumeSuspendCountPositive$
$C2\,ThreadResumeThreadStateSet$
$ThreadResume\,Outputs$

$return! = Kern\_success$

$RVThreadResumeSecureFailure$
$NotC1\,ThreadResumeSuspendCountPositive$
$C2\,ThreadResumeThreadStateSet$
$ThreadResume\,Outputs$

$return! = Kern\_failure$

$RVThreadResumeSecureInsufficientPermission$
$NotC2\,ThreadResumeThreadStateSet$
$ThreadResume\,Outputs$

$return! = Kern\_insufficient\_permission$

9.15.5 State Changes

A successful **thread_resume** or **thread_resume_secure** request decrements the thread's suspend count. If, as a result, the thread's suspend count becomes zero, the run state of the thread will be modified as follows. First, the thread will be taken out of the *Stopped* and *Halted* states. In addition, if the thread is not in the *Waiting* state, it will be placed in the *Running* state. The state *Uninterruptible* is not affected by this request. (Note that a thread may have the state *Uninterruptible* without having the state *Waiting*.) If the suspend count is not zero after it is decremented, the run state does not change. Nothing else changes due to the request.

$$
\begin{array}{l}
\textit{ThreadResumeInvariants} \\
\hline
\textit{ThreadInvariants} \\
\Xi\ \textit{ThreadExist} \\
\Xi\ \textit{TasksAndThreads} \\
\Xi\ \textit{ThreadPri} \\
\Xi\ \textit{ThreadSchedPolicy} \\
\Xi\ \textit{SpecialPurposePorts} \\
\Xi\ \textit{ThreadAndProcessorSet} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{ThreadResumeState} \\
\hline
\textit{ThreadResumeInvariants} \\
\Delta\ \textit{ThreadExecStatus} \\
\Delta\ \textit{Threads} \\
target\_thread? : THREAD \\
\hline
\underline{t}hread\_suspend\_count' = \underline{t}hread\_suspend\_count \\
\qquad \oplus \{target\_thread? \mapsto \underline{t}hread\_suspend\_count(target\_thread?) - 1\} \\
(\underline{t}hread\_suspend\_count'(target\_thread?) = 0 \land Waiting \notin \underline{r}un\_state(target\_thread?)) \\
\qquad \Rightarrow \underline{r}un\_state' = \underline{r}un\_state \\
\qquad\qquad \oplus \{target\_thread? \mapsto ((\underline{r}un\_state(target\_thread?) \setminus \{Stopped, Halted\}) \\
\qquad\qquad\qquad \cup \{Running\})\} \\
(\underline{t}hread\_suspend\_count'(target\_thread?) = 0 \land Waiting \in \underline{r}un\_state(target\_thread?)) \\
\qquad \Rightarrow \underline{r}un\_state' = \underline{r}un\_state \\
\qquad\qquad \oplus \{target\_thread? \mapsto (\underline{r}un\_state(target\_thread?) \setminus \{Stopped, Halted\})\} \\
\underline{t}hread\_suspend\_count'(target\_thread?) \neq 0 \Rightarrow \underline{r}un\_state' = \underline{r}un\_state \\
\underline{s}wapped\_threads' = \underline{s}wapped\_threads \\
\underline{i}dle\_threads' = \underline{i}dle\_threads \\
\underline{t}hreads\_wired' = \underline{t}hreads\_wired \\
\hline
\end{array}
$$

For the **thread_resume_secure** request the task creation state of the parent task is changed to Tcs_task_ready. There is no change to the task creation state of the parent task for a **thread_resume** request.

$$
\begin{array}{l}
\textit{ThreadResumeSecureState} \\
\hline
\Delta\ \textit{TaskCreationState} \\
\Xi\ \textit{TasksAndThreads} \\
target\_thread? : THREAD \\
operation? : OPERATION \\
\hline
(operation? = Thread\_resume\_secure\_id \\
\qquad \land\ target\_thread? \in \mathrm{dom}\ owning\_task \\
\qquad \land\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state \\
\qquad\qquad \oplus \{owning\_task(target\_thread?) \mapsto Tcs\_task\_ready\}) \\
\lor (operation? = Thread\_resume\_id \\
\qquad \land\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state) \\
\hline
\end{array}
$$

9.15.6 Complete Request

The following schemas define the general forms of the **thread_resume** and **thread_resume_secure** requests.

$$\begin{array}{l} \textit{ProcessingThreadResume} \\ \hline \textit{ProcessThreadViaThreadPortRequestGood} \\ \hline \textit{operation?} = \textit{Thread\_resume\_id} \end{array}$$

$$\begin{array}{l} \textit{ProcessingThreadResumeSecure} \\ \hline \textit{ProcessThreadViaThreadPortRequestGood} \\ \hline \textit{operation?} = \textit{Thread\_resume\_secure\_id} \end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$\begin{array}{l} \textit{ThreadResumeGood} \\ \quad \widehat{=} (\textit{RVThreadResumeGood} \land \textit{ThreadResumeState} \land \textit{ThreadResumeSecureState}) \\ \quad \gg \textit{RequestReturnOnlyStatus} \\ \textit{ThreadResumeSecureGood} \\ \quad \widehat{=} (\textit{RVThreadResumeSecureGood} \land \textit{ThreadResumeState} \land \textit{ThreadResumeSecureState}) \\ \quad \gg \textit{RequestReturnOnlyStatus} \end{array}$$

An unsuccessful request returns an error status.

$$\begin{array}{l} \textit{ThreadResumeBad} \mathrel{\widehat{=}} \textit{RVThreadResumeFailure} \gg \textit{RequestNoOp} \\ \textit{ThreadResumeSecureBad} \\ \quad \widehat{=} (\textit{RVThreadResumeSecureFailure} \lor \textit{RVThreadResumeSecureInsufficientPermission}) \\ \quad \gg \textit{RequestNoOp} \end{array}$$

Execution of the request consists of a good execution or an error execution.

$$\begin{array}{l} \textit{ExecuteThreadResume} \mathrel{\widehat{=}} \textit{ThreadResumeGood} \lor \textit{ThreadResumeBad} \\ \textit{ExecuteThreadResumeSecure} \mathrel{\widehat{=}} \textit{ThreadResumeSecureGood} \lor \textit{ThreadResumeSecureBad} \end{array}$$

The full specification for kernel processing of a validated **thread_resume** or **thread_resume_secure** request consists of processing the request followed by its execution.

$$\begin{array}{l} \textit{ThreadResume} \mathrel{\widehat{=}} \textit{ProcessingThreadResume} \fatsemi \textit{ExecuteThreadResume} \\ \textit{ThreadResumeSecure} \mathrel{\widehat{=}} \textit{ProcessingThreadResumeSecure} \fatsemi \textit{ExecuteThreadResumeSecure} \end{array}$$

9.16 thread_set_special_port

The **thread_set_special_port** request allows a task to set a specified special port for a specified thread to be the port associated with one of the task's send rights.

9.16.1 Client Interface

kern_return_t **thread_set_special_port**
(mach_port_t *thread_name*,
int *which_port*,
mach_port_t *special_port_name*);

thread_set_exception_port
Macro form

kern_return_t **thread_set_exception_port**
(mach_port_t *thread_name*,
mach_port_t *special_port_name*);

⇒ **thread_set_special_port** (*thread_name*, THREAD_EXCEPTION_PORT,
special_port_name)

thread_set_kernel_port
Macro form

kern_return_t **thread_set_kernel_port**
(mach_port_t *thread_name*,
mach_port_t *special_port_name*);

⇒ **thread_set_special_port** (*thread_name*, THREAD_KERNEL_PORT,
special_port_name)

9.16.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_set_special_port** request:

- $thread\_name?$ — the client's name for the thread whose special port is to be set
- $which\_port?$ — the type of special port that is to be set
- $special\_port\_name?$ — the client's name for the port to which the target thread's specified special port should be set

ThreadSetSpecialPortClientInputs

$thread\_name? : NAME$
$which\_port? : THREAD\_SPECIAL\_PORTS$
$special\_port\_name? : NAME$

A **thread_set_special_port** request is invoked by sending a message to the port indicated by $thread\_name?$ that has the operation field set to $Thread\_set\_special\_port\_id$ and has a body consisting of $which\_port?$ and $special\_port\_name?$.

$InvokeThreadSetSpecialPort$
$Invoke$
$ThreadSetSpecialPortClientInputs$

$trap\_id? = Mach\_msg\_trap$
$user\_spec?.message.header.operation = Thread\_set\_special\_port\_id$
$user\_spec?.message.header.remote\_port = thread\_name?$
$user\_spec?.message.body$
$\quad = Thread\_special\_port\_and\_name\_to\_text(which\_port?, special\_port\_name?)$

9.16.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_set_special_port** request:

- $return!$ — the status of the request

$ThreadSetSpecialPortClientOutputs$
$return! : KERNEL\_RETURN$

$ThreadSetSpecialPortReceiveReply$
$InvokeMachMsgRcv$
$ThreadSetSpecialPortClientOutputs$

$return! = Text\_to\_status(msg\_body)$

9.16.2 Kernel Interface

9.16.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_set_special_port** request:

- $thread?$ — the thread whose special port is to be set
- $which\_port?$ — the type of special port that is to be set
- $special\_port?$ — the port to which the target thread's specified special port should be set

$ThreadSetSpecialPortInputs$
$thread? : THREAD$
$which\_port? : THREAD\_SPECIAL\_PORTS$
$special\_port? : PORT$

9.16.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_set_special_port** request:

- $return!$ — the status of the request

$ThreadSetSpecialPortOutputs$
$return! : KERNEL\_RETURN$

9.16.3 Request Criteria

The following criteria are defined for the **thread_set_special_port** request.

- **C1 —** An exception port request is made.

$$\textit{C1ThreadSetExceptionPort}$$
$$which\_port? : THREAD\_SPECIAL\_PORTS$$
$$which\_port? = Thread\_exception\_port$$

$$NotC1ThreadSetExceptionPort \mathrel{\widehat{=}} \neg\ C1ThreadSetExceptionPort$$

- **C2 —** A kernel port request is made.

$$\textit{C2ThreadSetKernelPort}$$
$$which\_port? : THREAD\_SPECIAL\_PORTS$$
$$which\_port? = Thread\_kernel\_port$$

$$NotC2ThreadSetKernelPort \mathrel{\widehat{=}} \neg\ C2ThreadSetKernelPort$$

- **C3 —** The client has $Set\_thread\_exception\_port$ permission to the target thread.

Review Note:
In C3 and C4, we've begun the process of dealing with deferred permission checks under the new execution model. The first schema is used to initiate the permission checking routine and the criteria schemas will be used after the permission has been retrieved.

$$\textit{ThreadSetSpecialPortPermCheckSTEP}$$
$$Transition$$
$$\exists\, request : Request;\ CheckPending;\ ThreadSetSpecialPortInputs$$
$$\bullet\ curr\_bk?? = Bk\_have\_request(request)$$
$$\land\ request.operation = Thread\_set\_special\_port\_id$$
$$\land\ thread\_self(thread?) = request.service\_port$$
$$\land\ ssi = \underline{t}ask\_sid(curr\_task??)$$
$$\land\ osi = thread\_target(curr\_task??, thread?)$$
$$\land\ breaks' = breaks$$
$$\oplus \{\ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Set\_thread\_exception\_port, env)\}$$

$$\textit{C3ThreadCanSetExceptionPort}$$
$$Transition$$
$$env : ENVIRONMENT$$
$$curr\_bk?? = Bk\_have\_ruling(Set\_thread\_exception\_port, True, env)$$

$$\textit{NotC3ThreadCanSetExceptionPort}$$
$$Transition$$
$$env : ENVIRONMENT$$
$$curr\_bk?? = Bk\_have\_ruling(Set\_thread\_exception\_port, False, env)$$

- **C4 —** The client has $Set\_thread\_kernel\_port$ permission to the target thread.

$$
\begin{array}{l}
\textbf{ThreadSetSpecialPortPermCheckSTKP} \\
Transition \\
\hline
\exists\, request : Request;\ CheckPending;\ ThreadSetSpecialPortInputs \\
\bullet\ curr\_bk?? = Bk\_have\_request(request) \\
\quad \land\ request.operation = Thread\_set\_special\_port\_id \\
\quad \land\ thread\_self(thread?) = request.service\_port \\
\quad \land\ ssi = \underline{t}ask\_sid(curr\_task??) \\
\quad \land\ osi = thread\_target(curr\_task??, thread?) \\
\quad \land\ breaks' = breaks \\
\qquad \oplus \{ curr\_th?? \mapsto Bk\_check\_pending(ssi, osi, Set\_thread\_kernel\_port, env)\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{C4ThreadCanSetKernelPort} \\
Transition \\
env : ENVIRONMENT \\
\hline
curr\_bk?? = Bk\_have\_ruling(Set\_thread\_kernel\_port, True, env)
\end{array}
$$

$$
\begin{array}{l}
\textbf{NotC4ThreadCanSetKernelPort} \\
Transition \\
env : ENVIRONMENT \\
\hline
curr\_bk?? = Bk\_have\_ruling(Set\_thread\_kernel\_port, False, env)
\end{array}
$$

9.16.4 Return Values

Table 41 describes the values returned at the completion of the request and the conditions under which each value is returned. Note that C1 and C2 are mutually exclusive.

Review Note:
We assume that the prototype will check the conditions in the order {C1, C2 }, {C3 or C4}. However, the prototype is currently not checking C3 and C4.

$return$	C1	C2	C3	C4
$Kern\_success$	T	F	T	-
$Kern\_success$	F	T	-	T
$Kern\_insufficient\_permission$	T	F	F	-
$Kern\_insufficient\_permission$	F	T	-	F
$Kern\_invalid\_argument$	F	F	-	-

Table 41: Return Values for **thread_set_special_port**

$RVThreadSetExceptionPort$
$C1\,ThreadSetExceptionPort$
$NotC2\,ThreadSetKernelPort$
$C3\,ThreadCanSetExceptionPort$
$ThreadSetSpecialPort\,Outputs$

$return! = Kern\_success$

$RVThreadSetKernelPort$
$NotC1\,ThreadSetExceptionPort$
$C2\,ThreadSetKernelPort$
$C4\,ThreadCanSetKernelPort$
$ThreadSetSpecialPort\,Outputs$

$return! = Kern\_success$

$RVThreadCannotSetExceptionPort$
$C1\,ThreadSetExceptionPort$
$NotC2\,ThreadSetKernelPort$
$NotC3\,ThreadCanSetExceptionPort$
$ThreadSetSpecialPort\,Outputs$

$return! = Kern\_insufficient\_permission$

$RVThreadCannotSetKernelPort$
$NotC1\,ThreadSetExceptionPort$
$C2\,ThreadSetKernelPort$
$NotC4\,ThreadCanSetKernelPort$
$ThreadSetSpecialPort\,Outputs$

$return! = Kern\_insufficient\_permission$

$RVThreadSetSpecialPortInvalidPortType$
$NotC1\,ThreadSetExceptionPort$
$NotC2\,ThreadSetKernelPort$
$ThreadSetSpecialPort\,Outputs$

$return! = Kern\_invalid\_argument$

9.16.5 State Changes

A successful **thread_set_special_port** request sets the exception or kernel port of the thread to the given port.

$$
\begin{array}{l}
\textbf{ThreadSetExceptionPortState} \\
\hline
ThreadInvariants \\
\Xi\ Exist \\
\Xi\ Threads \\
\Xi\ PortNameSpace \\
\Delta\ SpecialPurposePorts \\
\Delta\ SpecialThreadPorts \\
\Xi\ ThreadAndProcessorSet \\
thread? : THREAD \\
special\_port? : PORT \\
\hline
thread\_self' = thread\_self \\
thread\_sself' = thread\_sself \\
thread\_eport' = thread\_eport \oplus \{thread? \mapsto special\_port?\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{ThreadSetKernelPortState} \\
\hline
ThreadInvariants \\
\Xi\ Exist \\
\Xi\ Threads \\
\Xi\ PortNameSpace \\
\Delta\ SpecialPurposePorts \\
\Delta\ SpecialThreadPorts \\
\Xi\ ThreadAndProcessorSet \\
thread? : THREAD \\
special\_port? : PORT \\
\hline
thread\_self' = thread\_self \\
thread\_sself' = thread\_sself \oplus \{thread? \mapsto special\_port?\} \\
thread\_eport' = thread\_eport
\end{array}
$$

Review Note:
The prototype also releases a send right on the former kernel or exception port. This can cause no-sender notifications to be sent if the number of send rights becomes zero. We have attempted to model the total number of send rights in $TotalSendRights$. However, we have not yet modeled the sending of notifications.

9.16.6 Complete Request

The general form of a **thread_set_special_port** request is

$$
\begin{array}{l}
\textbf{Processing ThreadSetSpecialPort} \\
\hline
ProcessThreadViaThreadPortRequestGood \\
\hline
operation? = Thread\_set\_special\_port\_id
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$\begin{aligned}&\mathit{ThreadSetSpecialPortGood}\\&\quad \widehat{=} ((\mathit{RVThreadSetExceptionPort} \land \mathit{ThreadSetExceptionPortState})\\&\qquad \lor (\mathit{RVThreadSetKernelPort} \land \mathit{ThreadSetKernelPortState}))\\&\quad \gg \mathit{ReturnOnlyStatus}\end{aligned}$$

An unsuccessful request returns an error status.

$$\begin{aligned}&\mathit{ThreadSetSpecialPortBad}\\&\quad \widehat{=} (\mathit{RVThreadCannotSetExceptionPort} \lor \mathit{RVThreadCannotSetKernelPort}\\&\qquad \lor \mathit{RVThreadSetSpecialPortInvalidPortType})\\&\quad \gg \mathit{NoOp}\end{aligned}$$

Execution of the request consists of a good execution or an error execution.

$$\mathit{ExecuteThreadSetSpecialPort} \mathrel{\widehat{=}} (\mathit{ThreadSetSpecialPortGood} \lor \mathit{ThreadSetSpecialPortBad})$$

The full specification for the kernel processing of a validated **thread_set_special_port** request consists of processing the request followed its execution.

$$\mathit{ThreadSetSpecialPort} \mathrel{\widehat{=}} \mathit{ProcessingThreadSetSpecialPort} \mathbin{⨾} \mathit{ExecuteThreadSetSpecialPort}$$

9.17 thread_set_state and thread_set_state_secure

The requests **thread_set_state** and **thread_set_state_secure** set the machine state of a specified thread. **thread_set_state_secure** can be used only if the thread was created using **thread_create_secure**.

9.17.1 Client Interface

```
kern_return_t thread_set_state
                (mach_port_t                 target_thread_name,
                 int                         flavor,
                 thread_state_t              new_state,
                 mach_msg_type_number_t      new_state_cnt);

kern_return_t thread_set_state_secure
                (mach_port_t                 target_thread_name,
                 int                         flavor,
                 thread_state_t              new_state,
                 mach_msg_type_number_t      new_state_cnt);
```

9.17.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_set_state** or **thread_set_state_secure** request:

- $\mathit{target\_thread\_name}?$ — the client's name for the thread whose state information is to be set

- $flavor?$ — the type of state information that is to be set
- $new\_state?$ — state information of the specified type for the target thread
- $new\_state\_cnt?$ — the maximum size that should be assumed for the state information supplied

$$
\begin{array}{l}
\textbf{ThreadSetStateClientInputs} \\
\hline
target\_thread\_name? : NAME \\
flavor? : THREAD\_STATE\_INFO\_TYPES \\
new\_state? : THREAD\_STATE\_INFO \\
new\_state\_cnt? : \mathbb{N} \\
\hline
\end{array}
$$

A **thread_set_state** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_set\_state\_id$ and has a body consisting of $flavor?$, $new\_state?$ and $new\_state\_cnt?$.

$$
\begin{array}{l}
\textbf{InvokeThreadSetState} \\
\hline
InvokeMachMsg \\
ThreadSetStateClientInputs \\
\hline
name? = target\_thread\_name? \\
operation? = Thread\_set\_state\_id \\
msg\_body = Thread\_set\_state\_params\_to\_text(flavor?, new\_state?, new\_state\_cnt?) \\
\hline
\end{array}
$$

A **thread_set_state_secure** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_set\_state\_secure\_id$ and has a body consisting of $flavor?$, $new\_state?$ and $new\_state\_cnt?$

$$
\begin{array}{l}
\textbf{InvokeThreadSetStateSecure} \\
\hline
InvokeMachMsg \\
ThreadSetStateClientInputs \\
\hline
name? = target\_thread\_name? \\
operation? = Thread\_set\_state\_secure\_id \\
msg\_body = Thread\_set\_state\_params\_to\_text(flavor?, new\_state?, new\_state\_cnt?) \\
\hline
\end{array}
$$

9.17.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_set_state** or **thread_set_state_secure** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textbf{ThreadSetStateClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{ThreadSetStateReceiveReply} \\
\hline
InvokeMachMsgRcv \\
ThreadSetStateClientOutputs \\
\hline
return! = Text\_to\_status(msg\_body) \\
\hline
\end{array}
$$

9.17.2 Kernel Interface

9.17.2.1 Input Parameters

The following input parameters are provided to the kernel for a **thread_set_state** or **thread_set_state_secure** request:

- $target\_thread?$ — the thread whose state information is to be set
- $flavor?$ — the type of state information that is to be set
- $new\_state?$ — state information of the specified type for the target thread
- $new\_state\_cnt?$ — the maximum size that should be assumed for the state information supplied

$$
\begin{array}{l}
\textit{ThreadSetStateInputs} \\
\hline
target\_thread? : THREAD \\
flavor? : THREAD\_STATE\_INFO\_TYPES \\
new\_state? : THREAD\_STATE\_INFO \\
new\_state\_cnt? : \mathbb{N} \\
\hline
\end{array}
$$

9.17.2.2 Output Parameters

The following output parameters are returned by the kernel for a **thread_set_state** or **thread_set_state_secure** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{ThreadSetStateOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

9.17.3 Request Criteria

The following criteria are defined for the **thread_set_state** and **thread_set_state_secure** requests.

- **C1 —** The parameter $target\_thread?$ is not equal to the client thread.

Editorial Note:
Nothing in the design states the reason that the client thread may not set its own state information. We believe that it is merely an implementation difficulty in that in order to set state information the thread must be temporarily stopped. If the client thread stopped itself, it could not set the state information.

$$
\begin{array}{l}
\textit{C1ThreadSetStateNotClientThread} \\
\hline
ThreadsAndProcessors \\
cpu?? : PROCESSOR \\
target\_thread? : THREAD \\
\hline
(cpu??, target\_thread?) \notin \underline{a}ctive\_thread \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC1ThreadSetStateNotClientThread \\
\quad \widehat{=}\ ThreadsAndProcessors \land \neg\ C1ThreadSetStateNotClientThread
\end{array}
$$

- **C2 —** The parameter $flavor?$ is a valid type of state information, and $new\_state\_cnt?$ is large enough for the requested state information.

$$
\begin{array}{|l}
\multicolumn{1}{l}{C2ThreadSetStateGoodFlavorAndCount} \\
\hline
ThreadMachineState \\
target\_thread? : THREAD \\
flavor? : THREAD\_STATE\_INFO\_TYPES \\
new\_state\_cnt? : \mathbb{N} \\
\hline
flavor? \in \mathrm{dom}\ Thread\_state\_count \\
\\
(target\_thread?, flavor?) \in \mathrm{dom}\ \underline{t}hread\_state \\
new\_state\_cnt? \geq Thread\_state\_count(flavor?) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC2ThreadSetStateGoodFlavorAndCount \\
\quad \widehat{=}\ ThreadMachineState \land \neg\ C2ThreadSetStateGoodFlavorAndCount
\end{array}
$$

- **C3 —** The task creation state of the target thread's owning task must be $Tcs\_thread\_created$. This criterion applies only to the **thread_set_state_secure** request.

$$
\begin{array}{|l}
\multicolumn{1}{l}{C3ThreadSetStateSecureThreadCreated} \\
\hline
TasksAndThreads \\
TaskCreationState \\
target\_thread? : THREAD \\
\hline
target\_thread? \in \mathrm{dom}\ owning\_task \\
owning\_task(target\_thread?) \in \mathrm{dom}\ \underline{t}ask\_creation\_state \\
\\
\underline{t}ask\_creation\_state(owning\_task(target\_thread?)) = Tcs\_thread\_created \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NotC3ThreadSetStateSecureThreadCreated \\
\quad \widehat{=}\ TasksAndThreads \land TaskCreationState \\
\quad \land \neg\ C3ThreadSetStateSecureThreadCreated
\end{array}
$$

9.17.4 Return Values

Table 42 describes the values returned at the completion of the **thread_set_state** request and the conditions under which each value is returned.

$return!$	C1	C2
$Kern\_success$	T	T
$Kern\_invalid\_argument$	otherwise	

Table 42: Return Values for **thread_set_state**

$$
\begin{array}{|l}
\multicolumn{1}{l}{RVThreadSetStateGood} \\
\hline
C1ThreadSetStateNotClientThread \\
C2ThreadSetStateGoodFlavorAndCount \\
ThreadSetStateOutputs \\
\hline
return! = Kern\_success \\
\hline
\end{array}
$$

$RVThreadSetStateInvalidFlavorOrCount$
$C1\ ThreadSetStateNotClientThread$
$NotC2\ ThreadSetStateGoodFlavorAndCount$
$ThreadSetState\ Outputs$

$return! = Kern\_invalid\_argument$

$RVThreadSetStateInvalidThread$
$NotC1\ ThreadSetStateNotClientThread$
$ThreadSetState\ Outputs$

$return! = Kern\_invalid\_argument$

Table 43 describes the values returned at the completion of the **thread_set_state_secure** request and the conditions under which each value is returned. In all cases where C1 is false we assume $Kern\_invalid\_argument$ is returned. In the case where C1 is true and both C2 and C3 are false we assume $Kern\_insufficient\_permission$ is returned.

Review Note:
In the prototype the order in which conditions are checked is C1, C3, C2.

$return!$	C1	C2	C3
$Kern\_success$	T	T	T
$Kern\_insufficient\_permission$	T	-	F
$Kern\_invalid\_argument$	otherwise		

Table 43: Return Values for **thread_set_state_secure**

$RVThreadSetStateSecureGood$
$C1\ ThreadSetStateNotClientThread$
$C2\ ThreadSetStateGoodFlavorAndCount$
$C3\ ThreadSetStateSecureThreadCreated$
$ThreadSetState\ Outputs$

$return! = Kern\_success$

$RVThreadSetStateSecureInsufficientPermission$
$C1\ ThreadSetStateNotClientThread$
$NotC3\ ThreadSetStateSecureThreadCreated$
$ThreadSetState\ Outputs$

$return! = Kern\_insufficient\_permission$

$RVThreadSetStateSecureInvalidFlavorOrCount$
$C1\ ThreadSetStateNotClientThread$
$NotC2\ ThreadSetStateGoodFlavorAndCount$
$C3\ ThreadSetStateSecureThreadCreated$
$ThreadSetState\ Outputs$

$return! = Kern\_invalid\_argument$

$$\begin{array}{|l} \hline RVThreadSetStateSecureInvalidThread \\ NotC1ThreadSetStateNotClientThread \\ ThreadSetStateOutputs \\ \hline return! = Kern\_invalid\_argument \\ \hline \end{array}$$

9.17.5 State Changes

A successful **thread_set_state** or **thread_set_state_secure** request sets the state of the thread to the supplied state information. Any information in $new\_state?$ in addition to that expected for $flavor?$ is ignored. The run state of the thread may also change since the request must ensure that the thread is temporarily suspended and then perhaps restart it.

$$\begin{array}{|l} \hline ThreadSetStateState \\ \Delta\ Threads \\ \Xi\ TasksAndThreads \\ \Xi\ ThreadPri \\ \Xi\ ThreadSchedPolicy \\ \Xi\ ThreadInstruction \\ \Delta\ ThreadExecStatus \\ \Xi\ ThreadStatistics \\ \Delta\ ThreadMachineState \\ \Xi\ Exist \\ \Xi\ SpecialPurposePorts \\ \Xi\ ThreadAndProcessorSet \\ ThreadInvariants \\ ThreadSetStateInputs \\ target\_thread? : THREAD \\ \hline \underline{t}hread\_state' = \underline{t}hread\_state \oplus \{(target\_thread?, flavor?) \mapsto new\_state?\} \\ ThreadDoWaitThenRelease[target\_thread?/stopping\_thread] \\ \underline{s}wapped\_threads' = \underline{s}wapped\_threads \\ \underline{i}dle\_threads' = \underline{i}dle\_threads \\ \underline{t}hread\_suspend\_count' = \underline{t}hread\_suspend\_count \\ \underline{t}hreads\_wired' = \underline{t}hreads\_wired \\ \hline \end{array}$$

$$\begin{array}{|l} \hline ThreadSetStateSecureState \\ \Delta\ TaskCreationState \\ \Xi\ TasksAndThreads \\ target\_thread? : THREAD \\ operation? : OPERATION \\ \hline (operation? = Thread\_set\_state\_secure\_id \\ \quad \wedge\ target\_thread? \in \mathrm{dom}\ owning\_task \\ \quad \wedge\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state \\ \qquad \oplus \{owning\_task(target\_thread?) \mapsto Tcs\_thread\_state\_set\}) \\ \vee\ (operation? = Thread\_set\_state\_id \\ \quad \wedge\ \underline{t}ask\_creation\_state' = \underline{t}ask\_creation\_state) \\ \hline \end{array}$$

9.17.6 Complete Request

The following schemas define the general forms of the **thread_set_state** and **thread_set_state_secure** requests.

$$\begin{array}{|l}
\textit{ProcessingThreadSetState} \\
\hline
\textit{ProcessThreadViaThreadPortRequestGood} \\
\hline
\textit{operation?} = \textit{Thread_set_state_id} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{ProcessingThreadSetStateSecure} \\
\hline
\textit{ProcessThreadViaThreadPortRequestGood} \\
\hline
\textit{operation?} = \textit{Thread_set_state_secure_id} \\
\hline
\end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$\begin{array}{l}
\textit{ThreadSetStateGood} \\
\quad \mathrel{\widehat=} (\textit{RVThreadSetStateGood} \land \textit{ThreadSetStateState} \\
\qquad \land\ \textit{ThreadSetStateSecureState}) \\
\quad \gg \textit{RequestReturnOnlyStatus} \\
\textit{ThreadSetStateSecureGood} \\
\quad \mathrel{\widehat=} (\textit{RVThreadSetStateSecureGood} \land \textit{ThreadSetStateState} \\
\qquad \land\ \textit{ThreadSetStateSecureState}) \\
\quad \gg \textit{RequestReturnOnlyStatus}
\end{array}$$

An unsuccessful request returns an error status.

$$\begin{array}{l}
\textit{ThreadSetStateBad} \\
\quad \mathrel{\widehat=} (\textit{RVThreadSetStateInvalidFlavorOrCount} \lor \textit{RVThreadSetStateInvalidThread}) \\
\quad \gg \textit{RequestNoOp} \\
\textit{ThreadSetStateSecureBad} \\
\quad \mathrel{\widehat=} (\textit{RVThreadSetStateSecureInvalidFlavorOrCount} \\
\qquad \lor\ \textit{RVThreadSetStateSecureInsufficientPermission} \\
\qquad \lor\ \textit{RVThreadSetStateSecureInvalidThread}) \\
\quad \gg \textit{RequestNoOp}
\end{array}$$

Execution of the request consists of a good execution or an error execution.

$$\begin{array}{l}
\textit{ExecuteThreadSetState} \mathrel{\widehat=} (\textit{ThreadSetStateGood} \lor \textit{ThreadSetStateBad}) \\
\textit{ExecuteThreadSetStateSecure} \mathrel{\widehat=} (\textit{ThreadSetStateSecureGood} \lor \textit{ThreadSetStateSecureBad})
\end{array}$$

The full specification for kernel processing of a validated **thread_set_state** or **thread_set_state_secure** request consists of processing the request followed by its execution.

$$\begin{array}{l}
\textit{ThreadSetState} \mathrel{\widehat=} \textit{ProcessingThreadSetState} \mathbin{;} \textit{ExecuteThreadSetState} \\
\textit{ThreadSetStateSecure} \mathrel{\widehat=} \textit{ProcessingThreadSetStateSecure} \mathbin{;} \textit{ExecuteThreadSetState}
\end{array}$$

9.18 thread_suspend

The request **thread_suspend** increments the suspend count of a thread by 1. If the thread was not already stopped, it will cause the thread to be stopped.

9.18.1 Client Interface

```
kern_return_t thread_suspend
        (mach_port_t                target_thread_name);
```

9.18.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_suspend** request:

- $target\_thread\_name?$ — the client's name for the thread that is to be suspended

```
── ThreadSuspendClientInputs ──────────────
│ target_thread_name? : NAME
└──────────────────────────────────────────
```

A **thread_suspend** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_suspend\_id$ and has no body.

```
── InvokeThreadSuspend ────────────────────
│ InvokeMachMsg
│ ThreadSuspendClientInputs
├──────────
│ name? = target_thread_name?
│ operation? = Thread_suspend_id
└──────────────────────────────────────────
```

9.18.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_suspend** request:

- $return!$ — the status of the request

```
── ThreadSuspendClientOutputs ─────────────
│ return! : KERNEL_RETURN
└──────────────────────────────────────────
```

```
── ThreadSuspendReceiveReply ──────────────
│ InvokeMachMsgRcv
│ ThreadSuspendClientOutputs
├──────────
│ return! = Text_to_status(msg_body)
└──────────────────────────────────────────
```

9.18.2 Kernel Interface

9.18.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_suspend** request:

- $target\_thread?$ — the thread that is to be suspended

$$\begin{array}{|l}\hline ThreadSuspendInputs \\ target\_thread? : THREAD \\ \hline\end{array}$$

9.18.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_suspend** request:

- $return!$ — the status of the request

$$\begin{array}{|l}\hline ThreadSuspendOutputs \\ return! : KERNEL\_RETURN \\ \hline\end{array}$$

9.18.3 Request Criteria

No criteria are defined for the **thread_suspend** request.

9.18.4 Return Values

Table 44 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$
$Kern\_success$

Table 44: Return Values for **thread_suspend**

$$\begin{array}{|l}\hline RVThreadSuspendGood \\ ThreadSuspendOutputs \\ \hline return! = Kern\_success \\ \hline\end{array}$$

9.18.5 State Changes

A successful **thread_suspend** request increments the thread's suspend count. The thread will obtain the run state of $Stopped$. (Note it is possible that the thread already has this state.) A thread in $Stopped$ status cannot execute any user level instructions or system traps. If a thread is suspending itself, then it will block (see Section 9.1.4.3). Otherwise, the run state $Running$ will be removed by $ThreadDoWait$ (see Section 9.1.4.3). The OSF documentation states that any system traps which are in progress when a thread is suspended will return after the thread resumes (via **thread_resume**).

ThreadSuspendState

$ThreadInvariants$
$\Xi\ ThreadExist$
$\Delta\ Threads$
$\Xi\ TasksAndThreads$
$\Xi\ ThreadPri$
$\Xi\ ThreadSchedPolicy$
$\Delta\ ThreadStatistics$
$\Delta\ ThreadExecStatus$
$\Xi\ Events$
$\Xi\ ThreadSampling$
$\Xi\ SpecialPurposePorts$
$\Xi\ ThreadAndProcessorSet$
$ThreadsAndProcessors$
$target\_thread? : THREAD$
$cpu?? : PROCESSOR$

$cpu?? \in \mathrm{dom}\ \underline{a}ctive\_thread$

$\underline{t}hreads\_wired' = \underline{t}hreads\_wired$
$\underline{t}hread\_suspend\_count' = \underline{t}hread\_suspend\_count$
$\quad \oplus \{target\_thread? \mapsto \underline{t}hread\_suspend\_count(target\_thread?) + 1\}$

$\mathbf{let}\ rs == \underline{r}un\_state \oplus \{target\_thread? \mapsto \underline{r}un\_state(target\_thread?) \cup \{Stopped\}\}$
$\bullet\ ((\underline{t}hread\_suspend\_count(target\_thread?) = 0$
$\quad \wedge\ target\_thread? = \underline{a}ctive\_thread(cpu??))$
$\qquad \Rightarrow ThreadBlock[target\_thread?/blocking\_thread, rs/init\_run\_state])$
$\wedge\ ((\underline{t}hread\_suspend\_count(target\_thread?) = 0$
$\quad \wedge\ target\_thread? \neq \underline{a}ctive\_thread(cpu??))$
$\qquad \Rightarrow ThreadDoWait[target\_thread?/stopping\_thread, rs/init\_run\_state])$
$\wedge\ (\underline{t}hread\_suspend\_count(target\_thread?) \neq 0$
$\quad \Rightarrow \underline{r}un\_state' = \underline{r}un\_state$
$\qquad \wedge\ \underline{s}wapped\_threads' = \underline{s}wapped\_threads$
$\qquad \wedge\ \underline{i}dle\_threads' = \underline{i}dle\_threads)$

Review Note:
The DTOS KID states that unpredictable results may occur if a program suspends a thread and alters its user state so that its direction is changed upon resuming.

9.18.6 Complete Request

The following schema defines the general form of **thread_suspend**.

ProcessingThreadSuspend

$ProcessThreadViaThreadPortRequestGood$

$operation? = Thread\_suspend\_id$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$ThreadSuspendGood \mathrel{\widehat=} (RVThreadSuspendGood \land ThreadSuspendState) \gg RequestReturnOnlyStatus$$

Execution of the request consists of a good execution.

$$ExecuteThreadSuspend \mathrel{\widehat=} ThreadSuspendGood$$

The full specification for kernel processing of a validated **thread_suspend** request consists of processing the request followed by its execution.

$$ThreadSuspend \mathrel{\widehat=} ProcessingThreadSuspend \fatsemi ExecuteThreadSuspend$$

9.19 thread_terminate

The request **thread_terminate** permanently stops execution of a thread.

9.19.1 Client Interface

```
kern_return_t thread_terminate
        (mach_port_t          target_thread_name);
```

9.19.1.1 Input Parameters

The following input parameters are provided by the client of a **thread_terminate** request:

- $target\_thread\_name?$ — the client's name for the thread to be destroyed

$$\begin{array}{|l}\hline ThreadTerminateClientInputs \\ target\_thread\_name? : NAME \\ \hline\end{array}$$

A **thread_terminate** request is invoked by sending a message to the port indicated by $target\_thread\_name?$ that has the operation field set to $Thread\_terminate\_id$ and has no body.

$$\begin{array}{|l}\hline InvokeThreadTerminate \\ InvokeMachMsg \\ ThreadTerminateClientInputs \\ \hline name? = target\_thread\_name? \\ operation? = Thread\_terminate\_id \\ \hline\end{array}$$

9.19.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **thread_terminate** request:

- $return!$ — the status of the request

ThreadTerminateClientOutputs

$return! : KERNEL\_RETURN$

ThreadTerminateReceiveReply

$InvokeMachMsgRcv$
$ThreadTerminateClientOutputs$

$return! = Text\_to\_status(msg\_body)$

9.19.2 Kernel Interface

9.19.2.1 Input Parameters The following input parameters are provided to the kernel for a **thread_terminate** request:

- $target\_thread?$ — the thread to be destroyed

ThreadTerminateInputs

$target\_thread? : THREAD$

9.19.2.2 Output Parameters The following output parameters are returned by the kernel for a **thread_terminate** request:

- $return!$ — the status of the request

ThreadTerminateOutputs

$return! : KERNEL\_RETURN$

9.19.3 Request Criteria

No criteria are defined for the **thread_terminate** request.

9.19.4 Return Values

Table 45 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$
$Kern\_success$

Table 45: Return Values for **thread_terminate**

Review Note:
It is actually possible for the prototype to return $Kern\_failure$. There are two cases where this appears to happen.

1. Someone else has already initiated a **thread_terminate** request on $target\_thread\_name?$. Although the current request returns $Kern\_failure$ and does not itself destroy the target thread, the thread is still destroyed by the other request which is in progress.
2. The client thread is currently being terminated itself. In this case, the client thread seems to hasten its own termination rather than finishing the current request. Thus, unless there were additional termination requests in progress for the target thread, it is *not* terminated.

It doesn't appear that our model is deep enough to handle either of these cases.

$$
\begin{array}{l}
\textbf{schema } RVThreadTerminateGood \\
\quad ThreadTerminateOutputs \\
\hline
\quad return! = Kern\_success
\end{array}
$$

9.19.5 State Changes

A successful **thread_terminate** request destroys the thread. The terminated thread is removed from the set of existing threads, and from its relationship with its parent task.

$$
\begin{array}{l}
\textbf{schema } ThreadTerminateStateExist \\
\quad \Delta\ ThreadExist \\
\quad \Delta\ TasksAndThreads \\
\quad target\_thread? : THREAD \\
\hline
\quad \underline{t}hread\_exists' = \underline{t}hread\_exists \setminus \{target\_thread?\} \\
\quad \underline{t}ask\_thread\_rel' = \underline{t}ask\_thread\_rel \rhd \{target\_thread?\}
\end{array}
$$

The processor assignment of the thread is also removed.

$$
\begin{array}{l}
\textbf{schema } ThreadTerminateStateThreadAndProcessorSet \\
\quad \Delta\ ThreadAndProcessorSet \\
\quad target\_thread? : THREAD \\
\hline
\quad \underline{t}hread\_assignment\_rel' = \{target\_thread?\} \lhd \underline{t}hread\_assignment\_rel \\
\quad \underline{e}nabled\_sp' = \underline{e}nabled\_sp \\
\quad \underline{p}s\_max\_priority' = \underline{p}s\_max\_priority
\end{array}
$$

The thread no longer has information associated with it regarding priorities, scheduling policies, statistics and sampling.

$$
\begin{array}{l}
\textbf{schema } ThreadTerminateStatePriority \\
\quad \Delta\ ThreadPri \\
\quad target\_thread? : THREAD \\
\hline
\quad \underline{t}hread\_priority' = \{target\_thread?\} \lhd \underline{t}hread\_priority \\
\quad \underline{t}hread\_max\_priority' = \{target\_thread?\} \lhd \underline{t}hread\_max\_priority \\
\quad \underline{t}hread\_sched\_priority' = \{target\_thread?\} \lhd \underline{t}hread\_sched\_priority \\
\quad \underline{d}epressed\_threads' = \underline{d}epressed\_threads \setminus \{target\_thread?\} \\
\quad \underline{p}riority\_before\_depression' = \{target\_thread?\} \lhd \underline{p}riority\_before\_depression
\end{array}
$$

$$
\begin{array}{l}
\textbf{ThreadTerminateStateSchedPolicy} \\
\hline
\Delta\ ThreadSchedPolicy \\
target\_thread? : THREAD \\
\hline
\underline{t}hread\_sched\_policy' = \{target\_thread?\} \lhd \underline{t}hread\_sched\_policy \\
\underline{t}hread\_sched\_policy\_data = \{target\_thread?\} \lhd \underline{t}hread\_sched\_policy\_data \\
\underline{s}upported\_sp' = \underline{s}upported\_sp
\end{array}
$$

$$
\begin{array}{l}
\textbf{ThreadTerminateStateStatistics} \\
\hline
\Delta\ ThreadStatistics \\
target\_thread? : THREAD \\
\hline
\underline{u}ser\_time' = \{target\_thread?\} \lhd \underline{u}ser\_time \\
\underline{s}ystem\_time' = \{target\_thread?\} \lhd \underline{s}ystem\_time \\
\underline{c}pu\_time' = \{target\_thread?\} \lhd \underline{c}pu\_time \\
\underline{s}leep\_time' = \{target\_thread?\} \lhd \underline{s}leep\_time
\end{array}
$$

$$
\begin{array}{l}
\textbf{ThreadTerminateStateSampling} \\
\hline
\Delta\ ThreadSampling \\
target\_thread? : THREAD \\
\hline
\underline{s}ampled\_threads' = \underline{s}ampled\_threads \setminus \{target\_thread?\} \\
\underline{t}hread\_sample\_types' = \{target\_thread?\} \lhd \underline{t}hread\_sample\_types \\
\underline{t}hread\_sample\_sequence\_number' \\
\quad = \{target\_thread?\} \lhd \underline{t}hread\_sample\_sequence\_number \\
\underline{t}hread\_samples' = \{target\_thread?\} \lhd \underline{t}hread\_samples
\end{array}
$$

The thread no longer has information associated with it regarding execution status. If the current thread is terminating itself, then it uses thread blocking to start another thread running.

Review Note:
The active thread on the CPU might change too. We have not modeled this change at all in the FTLS and say nothing about it here either.

$$
\begin{array}{l}
\textbf{ThreadTerminateStateExecStatus} \\
\hline
\Delta\ ThreadExecStatus \\
ThreadsAndProcessors \\
ProcessorAndProcessorSet \\
target\_thread? : THREAD \\
cpu?? : PROCESSOR \\
\hline
((cpu??, target\_thread?) \in \underline{a}ctive\_thread \\
\quad \Rightarrow (\mathbf{let}\ init\_run\_state == \{target\_thread?\} \lhd \underline{r}un\_state \\
\qquad \bullet\ ThreadBlock[target\_thread?/blocking\_thread])) \\
((cpu??, target\_thread?) \notin \underline{a}ctive\_thread \\
\quad \Rightarrow \underline{r}un\_state' = \{target\_thread?\} \lhd \underline{r}un\_state \\
\qquad \wedge\ \underline{s}wapped\_threads' = \underline{s}wapped\_threads \setminus \{target\_thread?\}) \\
\underline{t}hread\_suspend\_count' = \{target\_thread?\} \lhd \underline{t}hread\_suspend\_count \\
\underline{t}hreads\_wired' = \underline{t}hreads\_wired \setminus \{target\_thread?\}
\end{array}
$$

All special ports are removed from the thread.

Review Note:
The prototype releases send rights on the sself port and exception port. This can cause no-sender notifications to be sent if the number of send rights becomes zero. We have attempted to model the total number of send rights in $TotalSendRights$. However, we have not yet modeled the sending of notifications.

$$
\begin{array}{l}
\textbf{ThreadTerminateStateSpecialPorts} \\
\Delta\ SpecialPurposePorts \\
\Delta\ SpecialThreadPorts \\
target\_thread? : THREAD \\
\hline
thread\_self' = \{target\_thread?\} \lhd thread\_self \\
thread\_sself' = \{target\_thread?\} \lhd thread\_sself \\
thread\_eport' = \{target\_thread?\} \lhd thread\_eport
\end{array}
$$

The self port of the thread is destroyed.

$$
\begin{array}{l}
\textbf{ThreadTerminateStateSelfPort} \\
\Delta\ Ipc \\
SpecialThreadPorts \\
target\_thread? : THREAD \\
\hline
\mathbf{let}\ port == thread\_self(target\_thread?) \\
\bullet\ PortDestroy
\end{array}
$$

Any event for which the thread was waiting is disassociated from the thread. The corresponding event count is also incremented by 1.

Review Note:
I purposely violated the indentation conventions in this schema to show the nesting of the logical formulas.

$$
\begin{array}{l}
\textbf{ThreadTerminateStateEvent} \\
\Delta\ Events \\
target\_thread? : THREAD \\
\hline
\underline{t}hread\_waiting' = \underline{t}hread\_waiting \rhd \{target\_thread?\} \\
\underline{e}vent\_count' \\
\quad = \{event : EVENT\_COUNTER;\ count : \mathbb{N} \\
\qquad \mid event \in \mathrm{dom}\ \underline{e}vent\_count \\
\qquad \land (((event, target\_thread?) \notin \underline{t}hread\_waiting \\
\qquad\qquad \land\ count = \underline{e}vent\_count(event)) \\
\qquad\quad \lor ((event, target\_thread?) \in \underline{t}hread\_waiting \\
\qquad\qquad \land\ count = \underline{e}vent\_count(event) + 1)) \\
\qquad \bullet (event, count)\}
\end{array}
$$

State information is no longer available for the thread.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{ThreadTerminateStateMachineState} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta\ \mathit{ThreadMachineState} \\
\mathit{target\_thread?} : \mathit{THREAD} \\
\hline
\mathit{thread\_state}' \\
\quad = \{\, \mathit{info} : \mathit{THREAD\_STATE\_INFO\_TYPES} \\
\qquad \bullet (\mathit{target\_thread?}, \mathit{info}) \,\} \\
\quad \ntriangleleft \mathit{thread\_state} \\
\hline
\end{array}
$$

The thread no longer has an instruction pointer.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{ThreadTerminateStateInstr} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta\ \mathit{ThreadInstruction} \\
\mathit{target\_thread?} : \mathit{THREAD} \\
\hline
\mathit{instruction\_pointer}' = \{\mathit{target\_thread?}\} \ntriangleleft \mathit{instruction\_pointer} \\
\hline
\end{array}
$$

No other changes occur in the system state.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{ThreadTerminateState} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{ThreadInvariants} \\
\Delta\ \mathit{Threads} \\
\Xi\ \mathit{SpecialTaskPorts} \\
\mathit{ThreadTerminateStateExist} \\
\mathit{ThreadTerminateStatePriority} \\
\mathit{ThreadTerminateStateSchedPolicy} \\
\mathit{ThreadTerminateStateExecStatus} \\
\mathit{ThreadTerminateStateStatistics} \\
\mathit{ThreadTerminateStateSampling} \\
\mathit{ThreadTerminateStateSpecialPorts} \\
\mathit{ThreadTerminateStateThreadAndProcessorSet} \\
\mathit{ThreadTerminateStateEvent} \\
\mathit{ThreadTerminateStateMachineState} \\
\mathit{ThreadTerminateStateInstr} \\
\hline
\end{array}
$$

9.19.6 Complete Request

The following schema defines the general form of **thread_terminate**.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{ProcessingThreadTerminate} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{ProcessThreadViaThreadPortRequestGood} \\
\hline
\mathit{operation?} = \mathit{Thread\_terminate\_id} \\
\hline
\end{array}
$$

A request makes the state changes described in the previous section and creates a kernel reply.

$$
\begin{array}{l}
\mathit{ThreadTerminateGood} \mathrel{\widehat=} (\mathit{RVThreadTerminateGood} \land \mathit{ThreadTerminateState}) \\
\quad \gg \mathit{RequestReturnOnlyStatus}
\end{array}
$$

Execution of the request consists of a good execution.

$$
\mathit{ExecuteThreadTerminate} \mathrel{\widehat=} \mathit{ThreadTerminateGood}
$$

The full specification for kernel processing of a validated **thread_terminate** request consists of processing the request followed by its execution.

$$ThreadTerminate \mathrel{\widehat=} ProcessingThreadTerminate \mathbin{⨟} ExecuteThreadTerminate$$

Section 10 Virtual Memory Requests

10.1 Introduction to Virtual Memory Requests

This chapter describes the virtual memory kernel requests in DTOS.

10.1.1 Constants and Types

The following defines identifiers that are used to represent each of the requests. They are partitioned into Vm_task_ops and Vm_wire_id:

$$
\begin{array}{|l}
Vm\_allocate\_id, Vm\_allocate\_secure\_id, Vm\_copy\_id, Vm\_deallocate\_id, \\
\quad Vm\_inherit\_id, Vm\_machine\_attribute\_id, Vm\_map\_id, Vm\_protect\_id, \\
\quad Vm\_read\_id, Vm\_region\_id, Vm\_region\_secure\_id, Vm\_statistics\_id, \\
\quad Vm\_write\_id : OPERATION \\
Vm\_wire\_id : OPERATION \\
Vm\_task\_ops : \mathbb{P}\ OPERATION \\
\hline
\langle Vm\_allocate\_id, Vm\_allocate\_secure\_id, Vm\_copy\_id, Vm\_deallocate\_id, \\
\quad Vm\_inherit\_id, Vm\_machine\_attribute\_id, Vm\_map\_id, Vm\_protect\_id, \\
\quad Vm\_read\_id, Vm\_region\_id, Vm\_region\_secure\_id, Vm\_statistics\_id, Vm\_write\_id\rangle \\
\qquad Values\_partition\ Vm\_task\_ops \\
Vm\_task\_ops \subseteq Allowed\_mach\_services(Pc\_task) \\
Vm\_wire\_id \in Allowed\_mach\_services(Pc\_host\_control)
\end{array}
$$

10.1.2 Required Permissions

For each operation there is a primary permission that is required to perform the operation. We define here the portion of the $Required\_permission$ function that pertains to vm requests.

$$
\begin{array}{l}
\{(Vm\_allocate\_id, Allocate\_vm\_region), \\
\quad (Vm\_allocate\_secure\_id, Allocate\_vm\_region), \\
\quad (Vm\_copy\_id, Copy\_vm), \\
\quad (Vm\_deallocate\_id, Deallocate\_vm\_region), \\
\quad (Vm\_inherit\_id, Set\_vm\_region\_inherit), \\
\quad (Vm\_machine\_attribute\_id, Access\_machine\_attribute), \\
\quad (Vm\_map\_id, Map\_vm\_region), \\
\quad (Vm\_protect\_id, Chg\_vm\_region\_prot), \\
\quad (Vm\_read\_id, Read\_vm\_region), \\
\quad (Vm\_region\_id, Get\_vm\_region\_info), \\
\quad (Vm\_region\_secure\_id, Get\_vm\_region\_info), \\
\quad (Vm\_statistics\_id, Get\_vm\_statistics), \\
\quad (Vm\_write\_id, Write\_vm\_region)\} \\
\subset Required\_permission
\end{array}
$$

10.1.3 Invariant Information

No invariants are stated in this version of the VM Requests chapter.

10.1.4 General Information

10.1.4.1 Regions The following functions are needed to determine the pages specified by a request.

- $Get\_page(va)$ — determines the page index for the page of a virtual address va.
- $Get\_offset(va)$ — determines the offset on the page of a virtual address va.
- $Page\_start(va)$ — maps a virtual address va to the virtual address at the beginning of its page.
- $Address\_num(va)$ — maps a virtual address va to a number on which calculations can be performed.
- $Relative\_addr(addr, n)$ — calculates the address n bytes past the address $addr$ if such an address exists.
- $Page\_aligned$ — denotes the set of virtual addresses that are the beginning of a virtual page.

We assume that Vm_start and $Relative\_addr(Vm\_end, 1)$ are page aligned.

Review Note:
It might make sense to move these axioms and the $VAWord$ schema to the state chapter.

Editorial Note:
The definition of these functions as globals implies that there is a single global page size. This may not be true in a distributed environment with multiple processors of different types. The prototype uses a single global page size.

$$
\begin{array}{|l}
Get\_page : VIRTUAL\_ADDRESS \twoheadrightarrow PAGE\_INDEX \\
Get\_offset : VIRTUAL\_ADDRESS \twoheadrightarrow PAGE\_OFFSET \\
Page\_start : VIRTUAL\_ADDRESS \longrightarrow VIRTUAL\_ADDRESS \\
Page\_aligned : \mathbb{P}\ VIRTUAL\_ADDRESS \\
Address\_num : VIRTUAL\_ADDRESS \rightarrowtail \mathbb{N} \\
Relative\_addr : VIRTUAL\_ADDRESS \times \mathbb{N} \longrightarrow VIRTUAL\_ADDRESS \\
\hline
Page\_start \circ Page\_start = Page\_start \\
\forall va_1, va_2 : VIRTUAL\_ADDRESS \\
\bullet\ Get\_page(va_1) = Get\_page(va_2) \Leftrightarrow Page\_start(va_1) = Page\_start(va_2) \\
\quad \wedge\ va_1 \in Page\_aligned \Leftrightarrow va_1 = Page\_start(va_1) \\
\forall va_1, va_2 : VIRTUAL\_ADDRESS \\
\mid Get\_page(va_1) = Get\_page(va_2) \wedge Get\_offset(va_1) = Get\_offset(va_2) \\
\bullet\ va_1 = va_2 \\
\mathrm{dom}\ Relative\_addr = \{ addr : VIRTUAL\_ADDRESS; n : \mathbb{N} \\
\quad \mid Address\_num(addr) + n \in \mathrm{ran}\ Address\_num \} \\
\forall addr : VIRTUAL\_ADDRESS; n : \mathbb{N} \\
\mid (addr, n) \in \mathrm{dom}\ Relative\_addr \\
\bullet\ Relative\_addr(addr, n) = Address\_num^{\sim}(Address\_num(addr) + n) \\
Vm\_start \in Page\_aligned \\
Relative\_addr(Vm\_end, 1) \in Page\_aligned
\end{array}
$$

The contents of a task's address space at a particular virtual address is denoted by the function va_word.

$$
\begin{array}{|l}
\multicolumn{1}{l}{VAWord} \\
\hline
PageAndMemory \\
AddressSpace \\
va\_word : TASK \times VIRTUAL\_ADDRESS \longrightarrow WORD \\
\hline
\forall task : TASK; va : VIRTUAL\_ADDRESS \\
\mid (task, Get\_page(va)) \in \mathrm{dom}\ \underline{m}ap\_rel \\
\quad \wedge\ \underline{m}ap\_rel(task, Get\_page(va)) \in \mathrm{dom}\ representing\_page \\
\quad \wedge\ representing\_page(\underline{m}ap\_rel(task, Get\_page(va))) \in \mathrm{dom}\ page\_word\_fun \\
\bullet\ va\_word(task, va) \\
\quad = (page\_word\_fun(representing\_page(\underline{m}ap\_rel(task, \\
\qquad Get\_page(va)))))(Get\_offset(va)) \\
\hline
\end{array}
$$

We use $Region\_of(va, size)$ to denote the region of $size$ bytes starting at the page containing va in some task's address space. Since a region consists of a sequence of pages, the return from this function is the set of page indices denoting pages containing an address between va and $va + size - 1$. Because of this rounding to virtual page boundaries, the amount of memory in a region may be greater than $size$.

$$
\begin{array}{|l}
Region\_of : VIRTUAL\_ADDRESS \times \mathbb{N} \longrightarrow \mathbb{P}\ PAGE\_INDEX \\
\hline
\forall va : VIRTUAL\_ADDRESS; size : \mathbb{N} \\
\bullet\ Region\_of(va, size) = \{ va_1 : VIRTUAL\_ADDRESS \\
\quad \mid Address\_num(va_1) \in Address\_num(va) \mathrel{..} (Address\_num(va) + size - 1) \\
\quad \bullet\ Get\_page(va_1) \}
\end{array}
$$

$VmRegionInUse[task, address, size]$ denotes that $Region\_of(address, size)$ contains at least one page that is allocated in $task$'s address space, and $VmRegionNotInUse$ denotes that none of the pages are allocated.

$$
\begin{array}{l}
\textbf{VmRegionInUse} \\
\hline
AddressSpace \\
task : TASK \\
address : VIRTUAL\_ADDRESS \\
size : \mathbb{N} \\
\hline
Region\_of(address, size) \cap \underline{allocated}(\!|\{ task \}|\!) \neq \emptyset
\end{array}
$$

$$VmRegionNotInUse \mathrel{\widehat=} AddressSpace \land \neg\ VmRegionInUse$$

All addresses within a valid region must lie in the range $Vm\_start..Vm\_end$. We use $VmGoodRegion[address,size]$ to denote that the region of length $size$ starting at $address$ is valid.

Review Note:
Since we are assuming Vm_start and $Vm\_end + 1$ are page aligned we do not need to round $address$ and $size$.

$$
\begin{array}{l}
\textbf{VmGoodRegion} \\
\hline
address : VIRTUAL\_ADDRESS \\
size : \mathbb{N} \\
\hline
Address\_num(\ address\ ) \mathinner{..} Address\_num(\ address\ ) + size - 1 \\
\qquad \subseteq Address\_num(\ Vm\_start\ ) \mathinner{..} Address\_num(\ Vm\_end\ )
\end{array}
$$

Set_region_attr defines a function that maps all of the pages in a virtual memory region to a particular attribute.

$$
\begin{array}{l}
[R] \\
\hline
Set\_region\_attr : (\mathbb{P}(TASK \times PAGE\_INDEX) \times R) \\
\qquad\qquad \longrightarrow ((TASK \times PAGE\_INDEX) \mathrel{+\!\!\!\rightarrow} R) \\
\hline
\forall\ region : \mathbb{P}(TASK \times PAGE\_INDEX);\ x : R \\
\bullet\ Set\_region\_attr(region, x) = \\
\qquad \{\ task\_va\_pair : TASK \times PAGE\_INDEX \mid task\_va\_pair \in region \\
\qquad \bullet\ task\_va\_pair \mapsto x\ \}
\end{array}
$$

The **vm_write** request takes a vm map copy parameter that describes a region of virtual memory including the offset, the size and the task from whose address space the memory was copied. We model this with $MapCopy$.

Review Note:
In the prototype, a map copy does not contain a direct reference to the task. Although we are uncertain, it is even possible that the task from whose address space the map copy was produced no longer exists. It is conceivable that the task was destroyed after the map copy was created, and the map entries are still present since the map copy holds a reference to them. The correct solution here would be to model maps as entities in there own right independent of tasks. This would require significant changes to the state description.

$MapCopy$
$task : TASK$
$offset : VIRTUAL\_ADDRESS$
$size : \mathbb{N}$

10.1.4.2 Parameter Packaging Functions

When invoking a kernel request, the following functions package the parameters into a message body:

$$
\begin{aligned}
&Address\_to\_body : VIRTUAL\_ADDRESS \longrightarrow MESSAGE\_BODY\\
&Region\_to\_body : (VIRTUAL\_ADDRESS \times \mathbb{N}) \longrightarrow MESSAGE\_BODY\\
&Region\_bool\_to\_body : (VIRTUAL\_ADDRESS \times \mathbb{N} \times BOOLEAN)\\
&\quad \longrightarrow MESSAGE\_BODY\\
&Region\_inheritance\_to\_body :\\
&\quad (VIRTUAL\_ADDRESS \times \mathbb{N} \times INHERITANCE\_OPTION) \longrightarrow MESSAGE\_BODY\\
&Region\_bool\_sid\_to\_body :\\
&\quad (VIRTUAL\_ADDRESS \times \mathbb{N} \times BOOLEAN \times OSI) \longrightarrow MESSAGE\_BODY\\
&Region\_bool\_prot\_to\_body :\\
&\quad (VIRTUAL\_ADDRESS \times \mathbb{N} \times BOOLEAN \times \mathbb{P}\ PROTECTION)\\
&\qquad \longrightarrow MESSAGE\_BODY\\
&Address\_data\_to\_body :\\
&\quad (VIRTUAL\_ADDRESS \times MapCopy \times \mathbb{N}) \longrightarrow MESSAGE\_BODY\\
&Name\_region\_prot\_to\_body :\\
&\quad (NAME \times VIRTUAL\_ADDRESS \times \mathbb{N} \times \mathbb{P}\ PROTECTION) \longrightarrow MESSAGE\_BODY
\end{aligned}
$$

When creating a reply message from a request, the following functions package the output parameters into a kernel reply:

$$
\begin{aligned}
&Address\_to\_reply : VIRTUAL\_ADDRESS \longrightarrow KERNEL\_REPLY\\
&Attributes\_to\_reply : (VIRTUAL\_ADDRESS \times \mathbb{N} \times \mathbb{P}\ PROTECTION\\
&\qquad \times \mathbb{P}\ PROTECTION \times INHERITANCE\_OPTION\\
&\qquad \times BOOLEAN \times Capability \times OFFSET)\\
&\quad \longrightarrow KERNEL\_REPLY\\
&Secure\_attributes\_to\_reply : (VIRTUAL\_ADDRESS \times \mathbb{N} \times \mathbb{P}\ PROTECTION\\
&\qquad \times INHERITANCE\_OPTION \times BOOLEAN \times Capability\\
&\qquad \times \mathbb{P}\ PROTECTION \times OFFSET\\
&\qquad \times OSI \times \mathbb{P}\ Kernel\_permission)\\
&\quad \longrightarrow KERNEL\_REPLY
\end{aligned}
$$

When receiving a reply message from the kernel the following functions unpack the message body to obtain the output parameters (including the return status):

$$
\begin{aligned}
&Text\_to\_address\_and\_status : MESSAGE\_BODY \\
&\quad \longrightarrow (VIRTUAL\_ADDRESS \times KERNEL\_RETURN) \\
&Text\_to\_region\_info\_and\_status : MESSAGE\_BODY \\
&\quad \longrightarrow (VIRTUAL\_ADDRESS \times \mathbb{N} \times \mathbb{P}\ PROTECTION \\
&\qquad \times \mathbb{P}\ PROTECTION \times INHERITANCE\_OPTION \times BOOLEAN \\
&\qquad \times Capability \times OFFSET \times KERNEL\_RETURN) \\
&Text\_to\_region\_secure\_info\_and\_status : MESSAGE\_BODY \\
&\quad \longrightarrow (VIRTUAL\_ADDRESS \times \mathbb{N} \times \mathbb{P}\ PROTECTION \\
&\qquad \times \mathbb{P}\ PROTECTION \times INHERITANCE\_OPTION \times BOOLEAN \\
&\qquad \times Capability \times OFFSET \times \mathbb{P}\ PROTECTION \\
&\qquad \times OSI \times \mathbb{P}\ Kernel\_permission \times KERNEL\_RETURN)
\end{aligned}
$$

Review Note:
The command $Text\_to\_status$ is also used in this chapter. It is declared in the Thread Request chapter introduction.

10.1.5 Kernel Processing

The kernel performs processing for a VM request only when it detects a break indicating that a request has been received through a port of the appropriate class, Pc_task or $Pc\_host\_control$.

For a request sent to a task port, if the specified service port no longer exists, then a $Kern\_invalid\_argument$ status code is returned.

$NotTaskPort$

$$
\begin{aligned}
&ProcessRequest \\
&\Xi\ Mach \\
&reply\_to\_port! : \mathbb{P}\ PORT \\
&reply! : KERNEL\_REPLY \\
&return! : KERNEL\_RETURN
\end{aligned}
$$

$$
\begin{aligned}
&pc? = Pc\_task \\
&operation? \in Vm\_task\_ops \\
&service\_port? \notin \operatorname{dom} self\_task \\
&\\
&reply\_to\_port! = reply\_to\_port? \\
&return! = Kern\_invalid\_argument
\end{aligned}
$$

For a **vm_wire** request, which must be sent to a host control port, if the service port no longer exists, then a $Kern\_invalid\_host$ status code is returned.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ NotHostTaskPort \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
ProcessRequest \\
\Xi\ Mach \\
reply\_to\_port! : \mathbb{P}\ PORT \\
reply! : KERNEL\_REPLY \\
return! : KERNEL\_RETURN \\
\hline
pc? = Pc\_host\_control \\
operation? = Vm\_wire\_id \\
service\_port? \neq \underline{h}ost\_control\_port \\
\\
reply\_to\_port! = reply\_to\_port? \\
return! = Kern\_invalid\_host \\
\hline
\end{array}
$$

$$ProcessVMRequestBad \mathrel{\widehat=} (NotTaskPort \lor NotHostTaskPort) \gg RequestNoOp$$

Otherwise, the kernel processes the request. In this case, we use the following schema to represent the parameters to the requests:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ VMParameters \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
address? : VIRTUAL\_ADDRESS \\
anywhere? : BOOLEAN \\
copy? : BOOLEAN \\
count? : \mathbb{N} \\
cur\_protection? : \mathbb{P}\ PROTECTION \\
data? : MapCopy \\
data\_count? : \mathbb{N} \\
dest\_address? : VIRTUAL\_ADDRESS \\
host\_priv? : HOST \\
inheritance? : INHERITANCE\_OPTION \\
mask? : VIRTUAL\_ADDRESS \\
max\_protection? : \mathbb{P}\ PROTECTION \\
memory\_object? : Capability \\
new\_inheritance? : INHERITANCE\_OPTION \\
new\_protection? : \mathbb{P}\ PROTECTION \\
protection? : \mathbb{P}\ PROTECTION \\
obj\_sid? : OSI \\
offset? : OFFSET \\
set\_maximum? : BOOLEAN \\
shared? : BOOLEAN \\
size? : \mathbb{N} \\
source\_address? : VIRTUAL\_ADDRESS \\
target\_task? : TASK \\
wired\_access? : \mathbb{P}\ PROTECTION \\
\hline
\end{array}
$$

The interpretation of the components of this schema are:

address? — starting address for a region.

anywhere? — a Boolean indicating whether the region can be anywhere in the target task's address space.

$copy?$ — a Boolean indicating whether a copy is made of an area of a memory object.

$count?$ — the number of bytes in a region.

$cur\_protection?$ — the initial current protection for a region.

$data?$ — a copy of a portion of a memory map.

$data\_count?$ — the number of bytes in a data array (ignored).

$dest\_address?$ — starting address for the destination region.

$host\_priv?$ — the host on which the target task executes.

$inheritance?$ — the inheritance attribute for the region.

$mask?$ — alignment restrictions for the starting address of a region.

$max\_protection?$ — the maximum protection for a region.

$memory\_object?$ — the port naming a memory object.

$new\_inheritance?$ — the new inheritance attribute for the region.

$new\_protection?$ — the new protection for the region.

$protection?$ — the current protection for a region including those protections

$obj\_sid?$ — the security identifier for a region.

$offset?$ — an offset within a memory object, in bytes.

$set\_maximum?$ — a Boolean indicating whether the maximum protection or the current protection should be set.

$shared?$ — a Boolean indicating whether the region is shared with another task.

$size?$ — the number of bytes in a region.

$source\_address?$ — starting address for the source region.

$target\_task?$ — the task to whose address space the command applies.

$wired\_access?$ — the pageability of a region.

The following schema determines the target task based upon the task service port to which a task operation request has been sent.

$MessageToVMParameters$
$ProcessRequest$
$SpecialTaskPorts$
$VMParameters$

$pc? = Pc\_task$
$operation? \in Vm\_task\_ops$
$service\_port? \in \operatorname{dom} self\_task$
$target\_task? = self\_task(service\_port?)$

The following schema verifies that a **vm_wire** request has been sent to the host control port.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MessageToHostParameters}} \\
ProcessRequest \\
HostsAndPorts \\
VMParameters \\
\hline
pc? = Pc_host_control \\
operation? = Vm_wire_id \\
service_port? = \underline{h}ost_control_port
\end{array}$$

10.1.6 Security Server Request

For some requests (e.g., **vm_allocate**) a second security check is needed. In this case the access vector cache will be checked for the needed information. If the information is not present (or not valid for the client thread) the security server is queried, and the kernel must wait for the response before continuing the execution of the request. We represent this waiting time by adding an element to the set of pending requests that contains the current request, the client thread, and the OSI associated with the security server request. The schema $VmSecurityRequest$ checks the cache for permission $perm$ from the subject ssi to the object osi. If it is not found (i.e., $Cache\_undefined$), the request is placed in $PENDINGREQUEST$.

$$\begin{array}{|l}
Vm_request_to_pending_request : Request \times THREAD \times OSI \\
\quad \longrightarrow PENDREQUEST
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{VmSecurityRequest}} \\
Transition \\
KernelAllows \\
\Delta\ PendingRequests \\
ThreadsAndProcessors \\
Request? \\
perm : PERMISSION \\
ssi : SSI \\
osi : OSI \\
\hline
\mathbf{let}\ thread == \underline{a}ctive_thread(cpu??) \\
\bullet\ cache_allows(thread, ssi, osi, perm) = Cache_undefined \\
\quad \land PENDINGREQUEST' = PENDINGREQUEST \\
\qquad \uplus \llbracket Vm_request_to_pending_request(\theta Request?, thread, osi) \rrbracket
\end{array}$$

After the security request has been processed, the kernel request is removed from the set of pending requests by the schema $VmContinue$. The client thread and the OSI supplied in the security server request are also retrieved.

VmContinue

$$
\begin{array}{l}
\Delta PendingRequests \\
SpecialTaskPorts \\
VMParameters \\
Request? \\
thread' : THREAD \\
obj\_sid' : OSI \\
\hline
\exists pending\_request : PENDREQUEST \\
\bullet pending\_request \sqsubseteq PENDINGREQUEST \\
\quad \land pending\_request \\
\qquad = Vm\_request\_to\_pending\_request(\theta Request?, thread', obj\_sid') \\
\quad \land PENDINGREQUEST' = PENDINGREQUEST \uplus [\![pending\_request]\!] \\
\quad \land pc? = Pc\_task \\
\quad \land operation? \in Vm\_task\_ops
\end{array}
$$

If the required permission is already in the access vector cache, the security server request will not be necessary. *VmNoSecurityRequest* describes this case.

VmNoSecurityRequest

$$
\begin{array}{l}
Transition \\
KernelAllows \\
ThreadsAndProcessors \\
Request? \\
perm : PERMISSION \\
ssi : SSI \\
osi : OSI \\
\hline
\mathbf{let}\ thread == \underline{a}ctive\_thread(cpu??) \\
\bullet cache\_allows(thread, ssi, osi, perm) \neq Cache\_undefined
\end{array}
$$

We now describe the individual virtual memory requests.

10.2 vm_allocate and vm_allocate_secure

The **vm_allocate** and **vm_allocate_secure** task requests allocate a zero-filled region of memory in the target task's address space. The physical memory is not allocated until an executing thread references the new virtual memory, and a memory object managed by the default manager is not created until the region must be swapped out. **vm_allocate_secure** allows the client to specify a security identifier for the allocated region, while**vm_allocate** uses a default security identifier.

10.2.1 Client Interface

kern_return_t **vm_allocate**

(mach_port_t	*target_task_name,*
vm_address_t*	*address,*
vm_size_t	*size,*
boolean_t	*anywhere*);

```
kern_return_t vm_allocate_secure
              (mach_port_t        target_task_name,
               vm_address_t*      address,
               vm_size_t          size,
               boolean_t          anywhere,
               security_id_t      obj_sid);
```

10.2.1.1 Input Parameters The following input parameters are provided by the client of a **vm_allocate** request:

- $target\_task\_name?$ — the client's name for the task in whose virtual address space the region is to be allocated
- $address?$ — the requested starting address for the region. This parameter is ignored if $anywhere?$ is $True$. Otherwise, it is rounded down to the start of a page boundary.
- $size?$ — the number of bytes to allocate. It is rounded up to an integer number of pages. (This differs from the interpretation of $size?$ used in the other VM requests.)
- $anywhere?$ — a Boolean indicating whether the allocated region can be placed anywhere in the target task's address space or must be placed at $address?$

VmAllocateClientInputs

$target\_task\_name? : NAME$
$address? : VIRTUAL\_ADDRESS$
$size? : \mathbb{N}$
$anywhere? : BOOLEAN$

The following additional parameter must be provided by the client of a **vm_allocate_secure** request:

- $obj\_sid?$ — security identifier that will be attached to the newly allocated region.

VmAllocateSecureClientInputs

$VmAllocateClientInputs$
$obj\_sid? : OSI$

A **vm_allocate** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to $Vm\_allocate\_id$ and has a body consisting of $address?$, $size?$, and $anywhere?$.

InvokeVmAllocate

$InvokeMachMsg$
$VmAllocateClientInputs$

$name? = target\_task\_name?$
$operation? = Vm\_allocate\_id$
$msg\_body = Region\_bool\_to\_body\,(address?,\ size?,\ anywhere?)$

A **vm_allocate_secure** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to $Vm\_allocate\_secure\_id$ and has a body consisting of $address?$, $size?$, $anywhere?$, and $obj\_sid?$.

$$
\begin{array}{|l}
\hline
\textit{InvokeVmAllocateSecure} \\
\textit{InvokeMachMsg} \\
\textit{VmAllocateSecureClientInputs} \\
\hline
name? = target\_task\_name? \\
operation? = Vm\_allocate\_secure\_id \\
msg\_body = Region\_bool\_sid\_to\_body\,(address?,\ size?,\ anywhere?,\ obj\_sid?) \\
\hline
\end{array}
$$

10.2.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **vm_allocate** or **vm_allocate_secure** request:

- $address!$ — the actual starting address for the memory object
- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{VmAllocateClientOutputs} \\
address! : VIRTUAL\_ADDRESS \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{VmAllocateReceiveReply} \\
\textit{InvokeMachMsgRcv} \\
\textit{VmAllocateClientOutputs} \\
\hline
(address!, return!) = Text\_to\_address\_and\_status(msg\_body) \\
\hline
\end{array}
$$

10.2.2 Kernel Interface

10.2.2.1 Input Parameters The following input parameters are provided to the kernel for a **vm_allocate** request:

- $target\_task?$ — the task in whose virtual address space the region is to be allocated
- $address?$ — the requested starting address for the region. This parameter is ignored if $anywhere?$ is $True$. Otherwise, it is rounded down to the start of a page boundary.
- $size?$ — the number of bytes to allocate. It is rounded up to an integer number of pages. (This differs from the interpretation of $size?$ used in the other VM requests.)
- $anywhere?$ — a Boolean indicating whether the allocated region can be placed anywhere in the target task's address space or must be placed at $address?$

$$
\begin{array}{|l}
\hline
\textit{VmAllocateInputs} \\
target\_task? : TASK \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
anywhere? : BOOLEAN \\
\hline
\end{array}
$$

The following additional parameter must be provided by the client of a **vm_allocate_secure** request:

- $obj\_sid?$ — security identifier that will be attached to the newly allocated region.

$$
\begin{array}{l}
\textbf{VmAllocateSecureInputs} \\
\hline
VmAllocateInputs \\
obj\_sid? : OSI \\
\hline
\end{array}
$$

10.2.2.2 Output Parameters The following output parameters are returned by the kernel for a **vm_allocate** or **vm_allocate_secure** request:

- $address!$ — the actual starting address for the memory object
- $return!$ — the status of the request

$$
\begin{array}{l}
\textbf{VmAllocateOutputs} \\
\hline
address! : VIRTUAL\_ADDRESS \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

Upon completion of the processing of either a **vm_allocate** or a **vm_allocate_secure** request, a reply message is built from the output parameters.

$$
\begin{array}{l}
\textbf{VmAllocateReply} \\
\hline
RequestReturn \\
address? : VIRTUAL\_ADDRESS \\
\hline
reply? = Address\_to\_reply(\ address?\ ) \\
\hline
\end{array}
$$

10.2.3 Request Criteria

The following criteria are defined for the **vm_allocate** and **vm_allocate_secure** requests.

- **C1 —** The security identifier for the new region can be specified by the client thread, $\underline{a}ctive\_thread(cpu??)$, as determined from the result of a security policy query. The value of obj_sid is either $vm\_port\_sid(target\_task?)$ for a **vm_allocate** request or the input parameter $obj\_sid?$ for a **vm_allocate_secure** request. The binding of obj_sid is determined by the appropriate processing schema from Section 10.2.6.

$$
\begin{array}{l}
\textbf{C1\ VmAllocateGoodSecurityId} \\
\hline
SubjectSid \\
KernelAllows \\
ThreadsAndProcessors \\
thread : THREAD \\
obj\_sid : OSI \\
cpu?? : PROCESSOR \\
\hline
thread = \underline{a}ctive\_thread(cpu??) \\
thread \in \mathrm{dom}\ thread\_sid \\
cache\_allows(thread, thread\_sid(thread), obj\_sid, Map\_vm\_region) \\
\quad = Cache\_allowed \\
\hline
\end{array}
$$

$$NotC1VmAllocateGoodSecurityId \mathrel{\widehat=} SubjectSid \land KernelAllows \\ \land ThreadsAndProcessors \land \lnot C1VmAllocateGoodSecurityId$$

- **C2 —** The task remains after a possible second security server query has been made. The port $service\_port?$ is the port through which the request was received.

$$\begin{array}{l} \textbf{schema } C2VmAllocateTaskRemains \\ \quad SpecialTaskPorts \\ \quad target\_task? : TASK \\ \quad service\_port? : PORT \\ \hline \quad (service\_port?, target\_task?) \in self\_task \end{array}$$

$$NotC2VmAllocateTaskRemains \mathrel{\widehat=} SpecialTaskPorts \land \lnot C2VmAllocateTaskRemains$$

- **C3 —** The parameter $size?$ is greater than zero.

$$\begin{array}{l} \textbf{schema } C3VmAllocatePositiveSize \\ \quad size? : \mathbb{N} \\ \hline \quad size? > 0 \end{array}$$

$$NotC3VmAllocatePositiveSize \mathrel{\widehat=} \lnot C3VmAllocatePositiveSize$$

- **C4 —** The parameter $anywhere? = True$, or the addresses specified for the region (when rounded) are valid.

$$\begin{array}{l} \textbf{schema } C4VmAllocateGoodAddress \\ \quad address? : VIRTUAL\_ADDRESS \\ \quad size? : \mathbb{N} \\ \quad anywhere? : BOOLEAN \\ \hline \quad anywhere? = True \\ \quad \lor (\textbf{let } address == Page\_start(address?); size == size? \\ \qquad \bullet VmGoodRegion) \end{array}$$

$$NotC4VmAllocateGoodAddress \mathrel{\widehat=} \lnot C4VmAllocateGoodAddress$$

- **C5 —** There is room in $target\_task?$'s address space to allocate a region of length $size?$ starting at a page boundary. If $anywhere? = False$, there is room starting at the beginning of the page containing $address?$.

C5VmAllocateRoomToAllocate

$$
\begin{array}{l}
AddressSpace \\
target\_task? : TASK \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
anywhere? : BOOLEAN \\
address : VIRTUAL\_ADDRESS \\
\hline
\exists\, size : \mathbb{N} \\
\bullet\ (anywhere? = True \lor address = Page\_start(address?)) \\
\quad \land\ address \in Page\_aligned \\
\quad \land\ size = size? \\
\quad \land\ VmGoodRegion \\
\quad \land\ VmRegionNotInUse[target\_task?/task]
\end{array}
$$

$$NotC5VmAllocateRoomToAllocate \mathrel{\widehat{=}} AddressSpace \land \neg\, C5VmAllocateRoomToAllocate$$

Review Note:
Do we also require $Have\_execute$, $Have\_read$ and $Have\_write$ permissions for the target task?

10.2.4 Return Values

Table 46 describes the values returned at the completion of the request and the conditions under which each value is returned. The value $address$ is any address that satisfied the criterion C5. When $anywhere?$ is $False$, this is the address at the start of the page containing $address?$. When $anywhere?$ is $True$, the starting address of the allocated region depends upon $address?$, $size?$ and the allocated pages of $target\_task?$. The relationship between these three items and the address returned depends upon the implementation algorithm. In the prototype if C3 if false and C1 and C2 are true, the zero address, $Address\_num^{\sim}(0)$, is returned. We leave unspecified the precise address returned in cases where $return! \neq Kern\_success$.

Editorial Note:
The algorithm currently used in the prototype will never yield a page that starts earlier in the memory than the beginning of the page containing $address?$. Thus, if the client specifies the last page and it is already allocated, the return value will be $Kern\_no\_space$ even if there are pages available earlier in the address space.

The value of $address!$ when an error occurs is undefined in the design and therefore also depends on the implementation algorithm and is left unspecified. In the case where more than one error occurs we assume that the first applicable return status from the following list is returned: $Kern\_insufficient\_permission$, $Kern\_invalid\_argument$, $Kern\_invalid\_address$ and $Kern\_no\_space$.

Review Note:
The prototype checks the conditions in the order C1, C2, C3, C4 and C5.

$address!$	$return!$	C1	C2	C3	C4	C5
$address$	$Kern\_success$	T	T	T	T	T
—	$Kern\_no\_space$	T	T	T	T	F
—	$Kern\_invalid\_address$	T	T	T	F	-
$Address\_num\tilde{~}(0)$	$Kern\_success$	T	T	F	-	-
—	$Kern\_invalid\_argument$	T	F	-	-	-
—	$Kern\_insufficient\_permission$	F	-	-	-	-

Table 46: Return Values for **vm_allocate** and **vm_allocate_secure**

```
RVVmAllocateSuccessful
  AddressSpace
  VmAllocateOutputs
  C1VmAllocateGoodSecurityId
  C2VmAllocateTaskRemains
  C3VmAllocatePositiveSize
  C4VmAllocateGoodAddress
  C5VmAllocateRoomToAllocate
  ----
  address! = address
  return! = Kern_success
```

```
RVVmAllocateNoSpace
  AddressSpace
  VmAllocateOutputs
  C1VmAllocateGoodSecurityId
  C2VmAllocateTaskRemains
  C3VmAllocatePositiveSize
  C4VmAllocateGoodAddress
  NotC5VmAllocateRoomToAllocate
  ----
  return! = Kern_no_space
```

```
RVVmAllocateBadAddress
  AddressSpace
  VmAllocateOutputs
  C1VmAllocateGoodSecurityId
  C2VmAllocateTaskRemains
  C3VmAllocatePositiveSize
  NotC4VmAllocateGoodAddress
  ----
  return! = Kern_invalid_address
```

$$\begin{array}{|l}
\hline
\textit{RVVmAllocateVacuous} \\
AddressSpace \\
VmAllocateOutputs \\
C1\,VmAllocateGoodSecurityId \\
C2\,VmAllocateTaskRemains \\
NotC3\,VmAllocatePositiveSize \\
\hline
address! = Address_num^{\sim}(0) \\
return! = Kern_success \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{RVVmAllocateBadArgument} \\
AddressSpace \\
VmAllocateOutputs \\
C1\,VmAllocateGoodSecurityId \\
NotC2\,VmAllocateTaskRemains \\
\hline
return! = Kern_invalid_argument \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{RVVmAllocateBadSecurityId} \\
AddressSpace \\
VmAllocateOutputs \\
NotC1\,VmAllocateGoodSecurityId \\
\hline
return! = Kern_insufficient_permission \\
\hline
\end{array}$$

10.2.5 State Changes

When the request is successful, a new region $size?$ in length is added to the mapped address space for $target\_task?$ starting at $address!$ (one of the outputs calculated above). This region is initially mapped to the null memory object. The maximum protections for the new region are set so that they allow all accesses, and the current protections allow reading and writing. The $\underline{i}nheritance$ for the region is initialized to $Inheritance\_option\_copy$. The initial value of 0 will be set later when the region is first accessed.

Review Note:
The $\underline{c}\_protection'$ should take into account the access vector contents. It should be the intersection of read and write with the permissions allowed from the target task to its vm port sid.

$$
\begin{array}{l}
\textit{VmAllocateState} \\
\hline
\Delta\ AddressSpace \\
\Delta\ Protection \\
\Delta\ Inheritance \\
Memory \\
address! : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
target\_task? : TASK \\
\hline
\mathbf{let}\ region == \{ target\_task? \} \times Region\_of(address!, size?) \\
\bullet\ allocated' = allocated \cup region \\
\quad \land\ \underline{m}ap\_rel' (\!| region |\!) \subseteq (\{Null\_memory\} \times OFFSET) \\
\quad \land\ \underline{m}\_protection' = \underline{m}\_protection \\
\qquad \oplus Set\_region\_attr(region, \{Read, Write, Execute\}) \\
\quad \land\ \underline{c}\_protection' = \underline{c}\_protection \oplus Set\_region\_attr(region, \{Read, Write\}) \\
\quad \land\ \underline{i}nheritance' = \underline{i}nheritance \oplus Set\_region\_attr(region, Inheritance\_option\_copy)
\end{array}
$$

$$
\begin{array}{l}
\textit{VmAllocateSecureState} \\
\hline
\Delta\ PageSid \\
address! : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
target\_task? : TASK \\
obj\_sid : OSI \\
\hline
\underline{p}age\_sid' = \underline{p}age\_sid \\
\quad \oplus Set\_region\_attr((\{ target\_task? \} \times Region\_of(address!, size?)), obj\_sid)
\end{array}
$$

10.2.6 Complete Request

The general form of a **vm_allocate** request received through a task port has the following form. If a security server request is needed, then after the processing is begun the security request is made and the kernel request is marked as pending. It will later be continued by $VmContinue$. Note that obj_sid' is set to the default virtual memory security identifier for the target task.

$$
\begin{array}{l}
\textit{ProcessingVmAllocateSignature} \\
\hline
PortSid \\
MessageToVMParameters \\
\Delta\ DtosExec \\
\Xi\ Mach \\
\Xi\ DtosAdditions \\
\Xi\ ValidatedRequests \\
obj\_sid' : OSI \\
thread' : THREAD
\end{array}
$$

$$
\begin{array}{l}
\textit{ProcessingVmAllocateNoRequest} \\
\hline
\textit{Transition} \\
\textit{ProcessingVmAllocateSignature} \\
\hline
operation? = Vm\_allocate\_id \\
thread' = \underline{a}ctive\_thread(cpu??) \\
obj\_sid' = vm\_port\_sid(target\_task?) \\
\textbf{let } subject == thread\_sid(thread') \\
\bullet\ VmNoSecurityRequest[subject/ssi, obj\_sid'/osi, Map\_vm\_region/perm]
\end{array}
$$

$$
\begin{array}{l}
\textit{ProcessingVmAllocateWithRequest} \\
\hline
\textit{Transition} \\
\textit{ProcessingVmAllocateSignature} \\
\hline
operation? = Vm\_allocate\_id \\
thread' = \underline{a}ctive\_thread(cpu??) \\
obj\_sid' = vm\_port\_sid(target\_task?) \\
\textbf{let } subject == thread\_sid(thread') \\
\bullet\ VmSecurityRequest[subject/ssi, obj\_sid'/osi, Map\_vm\_region/perm]
\end{array}
$$

The general form of a **vm_allocate_secure** request received through a task port has the following form. If a security server request is needed, then after the processing is begun the security request is made and the kernel request is marked as pending. It will later be continued by $VmContinue$. Note that obj_sid' is set to the security identifier specified by $obj\_sid?$.

$$
\begin{array}{l}
\textit{ProcessingVmAllocateSecureNoRequest} \\
\hline
\textit{Transition} \\
\textit{ProcessingVmAllocateSignature} \\
\hline
operation? = Vm\_allocate\_secure\_id \\
thread' = \underline{a}ctive\_thread(cpu??) \\
obj\_sid' = obj\_sid? \\
\textbf{let } subject == thread\_sid(thread') \\
\bullet\ VmNoSecurityRequest[subject/ssi, obj\_sid'/osi, Map\_vm\_region/perm]
\end{array}
$$

$$
\begin{array}{l}
\textit{ProcessingVmAllocateSecureWithRequest} \\
\hline
\textit{Transition} \\
\textit{ProcessingVmAllocateSignature} \\
\hline
operation? = Vm\_allocate\_secure\_id \\
thread' = \underline{a}ctive\_thread(cpu??) \\
obj\_sid' = obj\_sid? \\
\textbf{let } subject == thread\_sid(thread') \\
\bullet\ VmSecurityRequest[subject/ssi, obj\_sid'/osi, Map\_vm\_region/perm]
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$
\begin{array}{l}
VmAllocateGood \\
\quad \widehat{=} ((RVVmAllocateSuccessful \lor RVVmAllocateVacuous) \\
\qquad \land VmAllocateState \land VmAllocateSecureState) \\
\quad \gg VmAllocateReply
\end{array}
$$

An unsuccessful request returns an error status.

$$
\begin{array}{l}
VmAllocateBad \\
\quad \widehat{=} \; (RVVmAllocateBadSecurityId \lor RVVmAllocateBadArgument \\
\qquad \lor RVVmAllocateBadAddress \lor RVVmAllocateNoSpace) \\
\quad \gg RequestNoOp
\end{array}
$$

Execution of the request consists of a good execution or an error execution.

Review Note:
The component $address$ is hidden so that $ExecuteVmAllocate$ has a signature consistent with other requests.

$$ExecuteVmAllocate \; \widehat{=} \; (VmAllocateGood \lor VmAllocateBad) \setminus (address)$$

The full specification for kernel processing of a validated **vm_allocate** or **vm_allocate_secure** request consists of processing the request, waiting until the correct information is in the access vector cache (if necessary), and then executing the request.

$$
\begin{array}{l}
VmAllocate \\
\quad \widehat{=} \; ([VmContinue \mid operation? = Vm\_allocate\_id] \lor ProcessingVmAllocateNoRequest) \\
\quad \fatsemi \; ExecuteVmAllocate \\
VmAllocateSecure \\
\quad \widehat{=} \; ([VmContinue \mid operation? = Vm\_allocate\_secure\_id] \\
\qquad \lor ProcessingVmAllocateSecureNoRequest) \\
\quad \fatsemi \; ExecuteVmAllocate
\end{array}
$$

10.3 vm_deallocate

The **vm_deallocate** task request deallocates a region of memory in the target task's address space.

10.3.1 Client Interface

```
kern_return_t vm_deallocate
                (mach_port_t        target_task_name,
                 vm_address_t       address,
                 vm_size_t          size);
```

10.3.1.1 Input Parameters

The following input parameters are provided by the client of a **vm_deallocate** request:

- $target\_task\_name?$ — the client's name for the task in whose virtual address space the region is to be deallocated
- $address?$ — starting address for the region

- $size?$ — the number of bytes to deallocate. Any page that contains an address in the range $address? \ldots (address? + size? - 1)$ will be deallocated.

$$\begin{array}{l} \textit{VmDeallocateClientInputs} \\ \hline target\_task\_name? : NAME \\ address? : VIRTUAL\_ADDRESS \\ size? : \mathbb{N} \\ \hline \end{array}$$

A **vm_deallocate** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to $Vm\_deallocate\_id$ and has a body consisting of $address?$ and $size?$.

$$\begin{array}{l} \textit{InvokeVmDeallocate} \\ \hline InvokeMachMsg \\ VmDeallocateClientInputs \\ \hline name? = target\_task\_name? \\ operation? = Vm\_deallocate\_id \\ msg\_body = Region\_to\_body\,(address?, size?) \\ \hline \end{array}$$

10.3.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **vm_deallocate** request:

- $return!$ — the status of the request

$$\begin{array}{l} \textit{VmDeallocateClientOutputs} \\ \hline return! : KERNEL\_RETURN \\ \hline \end{array}$$

$$\begin{array}{l} \textit{VmDeallocateReceiveReply} \\ \hline InvokeMachMsgRcv \\ VmDeallocateClientOutputs \\ \hline return! = Text\_to\_status(msg\_body) \\ \hline \end{array}$$

10.3.2 Kernel Interface

10.3.2.1 Input Parameters The following input parameters are provided to the kernel for a **vm_deallocate** request:

- $target\_task?$ — the task in whose virtual address space the region is to be deallocated
- $address?$ — starting address for the region
- $size?$ — the number of bytes to deallocate. Any page that contains an address in the range $address? \ldots (address? + size? - 1)$ will be deallocated.

$$\begin{array}{l} \textit{VmDeallocateInputs} \\ \hline target\_task? : TASK \\ address? : VIRTUAL\_ADDRESS \\ size? : \mathbb{N} \\ \hline \end{array}$$

10.3.2.2 Output Parameters

The following output parameters are returned by the kernel for a **vm_deallocate** request:

- $return!$ — the status of the request

$$\begin{array}{|l} \hline \textit{VmDeallocateOutputs} \\ return! : KERNEL\_RETURN \\ \hline \end{array}$$

10.3.3 Request Criteria

No criteria are defined for the **vm_deallocate** request.

10.3.4 Return Values

Table 47 describes the values returned at the completion of the request and the conditions under which each value is returned.

Editorial Note:
As noted by CLI, the OSF KID states that $Kern\_invalid\_address$ is returned if there are any unallocated pages in the region to be deallocated. The prototype always returns $Kern\_success$.

$return!$
$Kern\_success$

Table 47: Return Values for **vm_deallocate**

$$\begin{array}{|l} \hline \textit{RVVmDeallocateSuccessful} \\ VmDeallocateOutputs \\ \hline return! = Kern\_success \\ \hline \end{array}$$

10.3.5 State Changes

A successful **vm_deallocate** request deallocates virtual memory. It also deletes any system attributes that are only defined for allocated memory (protections, inheritance, security identifier).

$$
\begin{array}{l}
\textbf{schema } VmDeallocateState \\
\Delta\ AddressSpace \\
\Delta\ Protection \\
\Delta\ Inheritance \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
target\_task? : TASK \\
\hline
\textbf{let } region == \{ target\_task? \} \times Region\_of(address?, size?) \\
\bullet\ allocated' = allocated \setminus region \\
\quad \land\ \underline{m}ap\_rel' = region \lhd \underline{m}ap\_rel \\
\quad \land\ \underline{m}\_protection' = region \lhd \underline{m}\_protection \\
\quad \land\ \underline{c}\_protection' = region \lhd \underline{c}\_protection \\
\quad \land\ \underline{i}nheritance' = region \lhd \underline{i}nheritance
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } VmDeallocateSecureState \\
\Delta\ PageSid \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
target\_task? : TASK \\
\hline
\underline{p}age\_sid' = (\{ target\_task? \} \times Region\_of(address?, size?)) \lhd \underline{p}age\_sid
\end{array}
$$

10.3.6 Complete Request

The general form of a **vm_deallocate** request received through a task port has the following form.

$$
\begin{array}{l}
\textbf{schema } ProcessingVmDeallocate \\
MessageToVMParameters \\
\hline
operation? = Vm\_deallocate\_id
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$
\begin{array}{l}
VmDeallocateGood \mathrel{\widehat{=}} (RVVmDeallocateSuccessful \land VmDeallocateState) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

Execution of the request consists of a good execution.

$$ExecuteVmDeallocate \mathrel{\widehat{=}} VmDeallocateGood$$

The full specification for kernel processing of a validated **vm_deallocate** request consists of processing the request followed by its execution.

$$VmDeallocate \mathrel{\widehat{=}} ProcessingVmDeallocate \mathbin{;} ExecuteVmDeallocate$$

10.4 vm_inherit

The **vm_inherit** task request sets the inheritance attribute for a region within a specified task's address space.

10.4.1 Client Interface

```
kern_return_t vm_inherit
              (mach_port_t          target_task_name,
               vm_address_t         address,
               vm_size_t            size,
               vm_inherit_t         new_inheritance);
```

10.4.1.1 Input Parameters

The following input parameters are provided by the client of a **vm_inherit** request:

- $target\_task\_name?$ — the client's name for the task in whose virtual address space the region is contained
- $address?$ — starting address for the region
- $size?$ — the number of bytes in the region. The inheritance attributes will be modified for any page that contains an address in the range $address? \mathinner{\ldotp\ldotp} (address? + size? - 1)$.
- $new\_inheritance?$ — the new inheritance attribute for the region

$$
\begin{array}{l}
\textit{VmInheritClientInputs} \\
\hline
target\_task\_name? : NAME \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
new\_inheritance? : INHERITANCE\_OPTION \\
\hline
\end{array}
$$

A **vm_inherit** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to $Vm\_inherit\_id$ and has a body consisting of $address?$, $size?$, and $new\_inheritance?$.

$$
\begin{array}{l}
\textit{InvokeVmInherit} \\
\hline
InvokeMachMsg \\
VmInheritClientInputs \\
\hline
name? = target\_task\_name? \\
operation? = Vm\_inherit\_id \\
msg\_body = Region\_inheritance\_to\_body\,(address?, size?, new\_inheritance?) \\
\hline
\end{array}
$$

10.4.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **vm_inherit** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{VmInheritClientOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{_\ VmInheritReceiveReply\ ___} \\
InvokeMachMsgRcv \\
VmInheritClientOutputs \\
\hline
return! = Text_to_status(msg_body) \\
\hline
\end{array}$$

10.4.2 Kernel Interface

10.4.2.1 Input Parameters The following input parameters are provided to the kernel for a **vm_inherit** request:

- $target\_task?$ — the task in whose virtual address space the region is contained
- $address?$ — starting address for the region
- $size?$ — the number of bytes in the region. The inheritance attributes will be modified for any page that contains an address in the range $address? \mathrel{..} (address? + size? - 1)$.
- $new\_inheritance?$ — the new inheritance attribute for the region

$$\begin{array}{|l}
\multicolumn{1}{l}{_\ VmInheritInputs\ ___} \\
target_task? : TASK \\
address? : VIRTUAL_ADDRESS \\
size? : \mathbb{N} \\
new_inheritance? : INHERITANCE_OPTION \\
\hline
\end{array}$$

10.4.2.2 Output Parameters The following output parameters are returned by the kernel for a **vm_inherit** request:

- $return!$ — the status of the request

$$\begin{array}{|l}
\multicolumn{1}{l}{_\ VmInheritOutputs\ ___} \\
return! : KERNEL_RETURN \\
\hline
\end{array}$$

10.4.3 Request Criteria

The following criteria are defined for the **vm_inherit** request.

- **C1 —** The value of $new\_inheritance?$ is valid.

$$\begin{array}{|l}
\multicolumn{1}{l}{_\ C1VmInheritGoodInheritance\ ___} \\
new_inheritance? : INHERITANCE_OPTION \\
\hline
new_inheritance? \in \{ Inheritance_option_share, Inheritance_option_copy, Inheritance_option_none \} \\
\hline
\end{array}$$

$$NotC1VmInheritGoodInheritance \mathrel{\widehat=} \neg\ C1VmInheritGoodInheritance$$

10.4.4 Return Values

Table 48 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
Although the OSF KID states that $Kern\_invalid\_address$ is returned if the address is illegal or specifies a non-allocated region, in the prototype, $Kern\_invalid\_address$ is never returned for this request. It appears that $Kern\_success$ is returned in the case of a bad address. CLI has also noted this discrepancy.

$return!$	C1
$Kern\_success$	T
$Kern\_invalid\_argument$	F

Table 48: Return Values for **vm_inherit**

$RVVmInheritSuccessful$

$VmInheritOutputs$
$C1VmInheritGoodInheritance$

$return! = Kern\_success$

$RVVmInheritBadInheritance$

$VmInheritOutputs$
$NotC1VmInheritGoodInheritance$

$return! = Kern\_invalid\_argument$

10.4.5 State Changes

A successful **vm_inherit** sets the inheritance attribute for the region defined by $address?$ and $size?$ to the value specified by $new\_inheritance?$.

$VmInheritState$

$\Delta\ Inheritance$
$\Xi\ AddressSpace$
$target\_task? : TASK$
$address? : VIRTUAL\_ADDRESS$
$size? : \mathbb{N}$
$new\_inheritance? : INHERITANCE\_OPTION$

$$\mathbf{let}\ region == \{\ page\_index : PAGE\_INDEX \mid page\_index \in Region\_of(address?, size?) \land (target\_task?, page\_index) \in \underline{a}llocated \bullet (target\_task?, page\_index)\ \}$$
$$\bullet\ \underline{i}nheritance' = \underline{i}nheritance \oplus Set\_region\_attr(region, new\_inheritance?)$$

10.4.6 Complete Request

The general form of a **vm_inherit** request received through a task port has the following form.

$$\begin{array}{|l} \textit{ProcessingVmInherit} \\ \hline \textit{MessageToVMParameters} \\ \hline \textit{operation?} = \textit{Vm\_inherit\_id} \end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$VmInheritGood \mathrel{\widehat=} (RVVmInheritSuccessful \land VmInheritState) \\ \quad \gg RequestReturnOnlyStatus$$

An unsuccessful request returns an error status.

$$VmInheritBad \\ \quad \mathrel{\widehat=} RVVmInheritBadInheritance \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteVmInherit \mathrel{\widehat=} VmInheritGood \lor VmInheritBad$$

The full specification for kernel processing of a validated **vm_inherit** request consists of processing the request followed by its execution.

$$VmInherit \mathrel{\widehat=} ProcessingVmInherit \mathbin{;} ExecuteVmInherit$$

10.5 vm_protect

The **vm_protect** task request sets the current and/or maximum protections for a region within a specified task's address space. If the parameter $set\_maximum?$ is $False$, only the current protections are set. If $set\_maximum?$ is $True$, the maximum protections are set, and the current protections are also set so that they do not exceed the new maximum. Note that this request cannot be used to increase the maximum protections but only to decrease them.

10.5.1 Client Interface

```
kern_return_t vm_protect
        (mach_port_t        target_task_name,
         vm_address_t       address,
         vm_size_t          size,
         boolean_t          set_maximum,
         vm_prot_t          new_protection);
```

10.5.1.1 Input Parameters

The following input parameters are provided by the client of a **vm_protect** request:

- $target\_task\_name?$ — the client's name for the task in whose virtual address space the region is contained
- $address?$ — starting address for the region
- $size?$ — the number of bytes in the region. The protections will be modified for any page that contains an address in the range $address? \mathinner{\ldotp\ldotp} (address? + size? - 1)$.
- $set\_maximum?$ — a Boolean indicating whether the maximum protection should be set. A value of $True$ indicates the maximum protection should be set. (The current protection is also set if it violates the new maximum.) A value of $False$ indicates only the current protection is set.
- $new\_protection?$ — the new protection for the region

$VmProtectClientInputs$
$$\begin{array}{l} target\_task\_name? : NAME \\ address? : VIRTUAL\_ADDRESS \\ size? : \mathbb{N} \\ set\_maximum? : BOOLEAN \\ new\_protection? : \mathbb{P}\ PROTECTION \end{array}$$

A **vm_protect** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to $Vm\_protect\_id$ and has a body consisting of $address?$, $size?$, $set\_maximum?$, and $new\_protection?$.

$InvokeVmProtect$
$$\begin{array}{l} InvokeMachMsg \\ VmProtectClientInputs \\ \hline name? = target\_task\_name? \\ operation? = Vm\_protect\_id \\ msg\_body \\ \quad = Region\_bool\_prot\_to\_body(\ address?, size?, set\_maximum?, new\_protection?\ ) \end{array}$$

10.5.1.2 Output Parameters

The following output parameters are received through the reply port provided by the client of a **vm_protect** request:

- $return!$ — the status of the request

$VmProtectClientOutputs$
$$return! : KERNEL\_RETURN$$

$VmProtectReceiveReply$
$$\begin{array}{l} InvokeMachMsgRcv \\ VmProtectClientOutputs \\ \hline return! = Text\_to\_status(msg\_body) \end{array}$$

10.5.2 Kernel Interface

10.5.2.1 Input Parameters The following input parameters are provided to the kernel for a **vm_protect** request:

- $target\_task?$ — the task in whose virtual address space the region is contained
- $address?$ — starting address for the region
- $size?$ — the number of bytes in the region. The protections will be modified for any page that contains an address in the range $address? \mathrel{..} (address? + size? - 1)$.
- $set\_maximum?$ — a Boolean indicating whether the maximum protection should be set. A value of $True$ indicates the maximum protection should be set. (The current protection is also set if it violates the new maximum.) A value of $False$ indicates only the current protection is set.
- $new\_protection?$ — the new protection for the region

$$
\begin{array}{l}
\textit{VmProtectInputs} \\
\hline
target\_task? : TASK \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
set\_maximum? : BOOLEAN \\
new\_protection? : \mathbb{P}\ PROTECTION \\
\hline
\end{array}
$$

10.5.2.2 Output Parameters The following output parameters are returned by the kernel for a **vm_protect** request:

- $return!$ — the status of the request

$$
\begin{array}{l}
\textit{VmProtectOutputs} \\
\hline
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

10.5.3 Request Criteria

The following criteria are defined for the **vm_protect** request.

- **C1 —** The new protection is less than the existing maximum protection.

$$
\begin{array}{l}
\textit{C1 VmProtectGoodProtection} \\
\hline
Protection \\
target\_task? : TASK \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
new\_protection? : \mathbb{P}\ PROTECTION \\
\hline
\forall\, page\_index : PAGE\_INDEX \\
\mid page\_index \in Region\_of(address?, size?) \\
\bullet\ (target\_task?, page\_index) \in \operatorname{dom} \underline{m}\_protection \\
\quad \wedge\ new\_protection? \subseteq \underline{m}\_protection\,(target\_task?, page\_index) \\
\hline
\end{array}
$$

$$NotC1\,VmProtectGoodProtection \mathrel{\widehat=} Protection \land \lnot\ C1\,VmProtectGoodProtection$$

10.5.4 Return Values

Table 49 describes the values returned at the completion of the request and the conditions under which each value is returned.

Review Note:
Although the OSF KID states that $Kern\_invalid\_address$ is returned if the address is illegal or specifies a non-allocated region, in the prototype, $Kern\_invalid\_address$ is never returned for this request. It appears that $Kern\_success$ is returned in the case of an unallocated page. CLI has also noted this discrepancy.

$return!$	C1
$Kern\_success$	T
$Kern\_protection\_failure$	F

Table 49: Return Values for **vm_protect**

$RVVmProtectSuccessful$
$VmProtectOutputs$
$C1\,VmProtectGoodProtection$

$return! = Kern\_success$

$RVVmProtectBadProtection$
$VmProtectOutputs$
$NotC1\,VmProtectGoodProtection$

$return! = Kern\_protection\_failure$

10.5.5 State Changes

A successful **vm_protect** sets either the maximum or the current memory protection (read, write, and/or execute) allowed for the region, depending on whether $set\_maximum?$ is $True$ or $False$. If the maximum is set below the current protection, the current protection must also be adjusted to remove any permissions that are not within the new maximum.

$$
\begin{array}{l}
\textit{VmProtectState} \\
\hline
\Delta\ Protection \\
\Xi\ AddressSpace \\
target\_task? : TASK \\
address? : VIRTUAL\_ADDRESS \\
size? : \mathbb{N} \\
set\_maximum? : BOOLEAN \\
new\_protection? : \mathbb{P}\ PROTECTION \\
\hline
\textbf{let}\ region == \{\ page\_index : PAGE\_INDEX \\
\quad \mid page\_index \in Region\_of(address?, size?) \\
\qquad \wedge (target\_task?, page\_index) \in allocated \\
\quad \bullet (target\_task?, page\_index)\ \} \\
\bullet\ (set\_maximum? = True \\
\quad \wedge\ \underline{m}\_protection' = \underline{m}\_protection \oplus Set\_region\_attr(region, new\_protection?) \\
\quad \wedge\ \underline{c}\_protection' = \underline{c}\_protection \\
\qquad \oplus \{\ task\_va\_pair : TASK \times PAGE\_INDEX \mid task\_va\_pair \in region \\
\qquad\quad \bullet\ task\_va\_pair \mapsto \underline{c}\_protection(task\_va\_pair) \cap new\_protection?\ \}) \\
\vee\ (set\_maximum? = False \\
\quad \wedge\ \underline{c}\_protection' = \underline{c}\_protection \oplus Set\_region\_attr(region, new\_protection?))
\end{array}
$$

Review Note:
This ignores the protections coming from the security server. Right now these protections are not in the model of the state.

10.5.6 Complete Request

The general form of a **vm_protect** request received through a task port has the following form.

$$
\begin{array}{l}
\textit{ProcessingVmProtect} \\
\hline
MessageToVMParameters \\
\hline
operation? = Vm\_protect\_id
\end{array}
$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$
\begin{array}{l}
VmProtectGood \mathrel{\widehat=} (RVVmProtectSuccessful \wedge VmProtectState) \\
\quad \gg RequestReturnOnlyStatus
\end{array}
$$

An unsuccessful request returns an error status.

$$
\begin{array}{l}
VmProtectBad \\
\quad \mathrel{\widehat=} RVVmProtectBadProtection \gg RequestNoOp
\end{array}
$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteVmProtect \mathrel{\widehat=} VmProtectGood \vee VmProtectBad$$

The full specification for kernel processing of a validated **vm_protect** request consists of processing the request followed by its execution.

$$VmProtect \mathrel{\widehat=} ProcessingVmProtect \fatsemi ExecuteVmProtect$$

10.6 vm_write

The **vm_write** task request writes an allocated region in the target task's address space.

10.6.1 Client Interface

```
kern_return_t vm_write
        (mach_port_t                  target_task_name,
         vm_address_t                 address,
         vm_offset_t                  data,
         mach_msg_type_number_t       data_count);
```

10.6.1.1 Input Parameters

The following input parameters are provided by the client of a **vm_write** request:

- $target\_task\_name?$ — the client's name for the task in whose virtual address space the region is to be written
- $address?$ — starting address for the destination region, which must be the start of a page boundary
- $data?$ — the data to be written

> *Editorial Note:*
> In the DTOS KID, this parameter is described as a page-aligned array of data. However, in the prototype the $data?$ parameter is a pointer to a vm_map_copy structure which encodes information about the source region to copy including its offset, size, a type and a memory map. The type describes how this structure represents the data. The three possibilities are an entry list, an object and a page list (only entry lists are currently supported by the prototype). This structure is returned by a **vm_read** request. We will model this structure as a $MapCopy$ containing the offset, the size and the task from whose address space the copy was made.

- $data\_count?$ — ignored

> *Editorial Note:*
> In the DTOS KID this parameter denotes the number of bytes in the array pointed to by the $data?$ parameter. However, the number of bytes is included in the vm_map_copy structure, and the $data\_count?$ parameter is ignored in the prototype.

$VmWriteClientInputs$

$target\_task\_name? : NAME$
$address? : VIRTUAL\_ADDRESS$
$data? : MapCopy$
$data\_count? : \mathbb{N}$

A **vm_write** request is invoked by sending a message to the port indicated by $target\_task\_name?$ that has the operation field set to Vm_write_id and has a body consisting of $address?$, $data?$, and $data\_count?$.

$$
\begin{array}{|l}
\hline
\textit{InvokeVmWrite} \\
\textit{InvokeMachMsg} \\
\textit{VmWriteClientInputs} \\
\hline
name? = target\_task\_name? \\
operation? = Vm\_write\_id \\
msg\_body = Address\_data\_to\_body\,(address?,\ data?,\ data\_count?) \\
\hline
\end{array}
$$

10.6.1.2 Output Parameters The following output parameters are received through the reply port provided by the client of a **vm_write** request:

- $return!$ — the status of the request

$$
\begin{array}{|l}
\hline
\textit{VmWriteClientOutputs} \\
return! : KERNEL\_RETURN \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{VmWriteReceiveReply} \\
\textit{InvokeMachMsgRcv} \\
\textit{VmWriteClientOutputs} \\
\hline
return! = Text\_to\_status(msg\_body) \\
\hline
\end{array}
$$

10.6.2 Kernel Interface

10.6.2.1 Input Parameters The following input parameters are provided to the kernel for a **vm_write** request:

- $target\_task?$ — the task in whose virtual address space the region is to be written
- $address?$ — starting address for the destination region, which must be the start of a page boundary
- $data?$ — the data to be written

Editorial Note:
In the DTOS KID, this parameter is described as a page-aligned array of data. However, in the prototype the $data?$ parameter is a pointer to a vm_map_copy structure which encodes information about the source region to copy including its offset, size, a type and a memory map. The type describes how this structure represents the data. The three possibilities are an entry list, an object and a page list (only entry lists are currently supported by the prototype). This structure is returned by a **vm_read** request. We will model this structure as a $MapCopy$ containing the offset, the size and the task from whose address space the copy was made.

- $data\_count?$ — ignored

Editorial Note:
In the DTOS KID this parameter denotes the number of bytes in the array pointed to by the $data?$ parameter. However, the number of bytes is included in the vm_map_copy structure, and the $data\_count?$ parameter is ignored in the prototype.

$$\text{VmWriteInputs}$$
$$target\_task? : TASK$$
$$address? : VIRTUAL\_ADDRESS$$
$$data? : MapCopy$$
$$data\_count? : \mathbb{N}$$

10.6.2.2 Output Parameters The following output parameters are returned by the kernel for a **vm_write** request:

- $return!$ — the status of the request

$$\text{VmWriteOutputs}$$
$$return! : KERNEL\_RETURN$$

10.6.3 Request Criteria

The following criteria are defined for the **vm_write** request.

- **C1 —** The parameter $address?$ and the offset included in the parameter $data?$ are on a page boundary. Also, the size included in $data?$ is an integer number of pages.

$$\text{C1VmWritePageAligned}$$
$$address? : VIRTUAL\_ADDRESS$$
$$data? : MapCopy$$
$$\{address?, data?.offset, Address\_num^{\sim}(data?.size)\} \subseteq Page\_aligned$$

$$NotC1VmWritePageAligned \mathrel{\widehat=} \neg\ C1VmWritePageAligned$$

- **C2 —** The addresses specified for the destination region are valid and are allocated.

$$\text{C2VmWriteGoodAddress}$$
$$AddressSpace$$
$$target\_task? : TASK$$
$$address? : VIRTUAL\_ADDRESS$$
$$data? : MapCopy$$
$$\mathbf{let}\ data\_size == data?.size$$
$$\bullet\ VmGoodRegion[address?/address, data\_size/size]$$
$$\quad \wedge\ Region\_of(address?, data\_size) \subseteq \underline{a}llocated(\!|\{target\_task?\}|\!)$$

$$NotC2VmWriteGoodAddress \mathrel{\widehat=} AddressSpace \wedge \neg\ C2VmWriteGoodAddress$$

- **C3 —** The target task has permission to write to the region.

Review Note:
I believe the security server should be queried to make sure the target task still has write permission. However, I don't think the prototype currently makes this check (11/15/94).

$C3\,VmWriteWritable$
$Protection$
$target\_task? : TASK$
$address? : VIRTUAL\_ADDRESS$
$data? : MapCopy$

$\forall\, page : PAGE\_INDEX$
$\mid page \in Region\_of(address?, data?.size)$
$\quad \wedge (target\_task?, page) \in \mathrm{dom}\ \underline{c}\_protection$
$\bullet\ Write \in \underline{c}\_protection(\, target\_task?, page\,)$

$$NotC3\,VmWriteWritable \mathrel{\widehat{=}} Protection \wedge \neg\ C3\,VmWriteWritable$$

Review Note:
Should we state that read permission is required on the source region? The prototype does require this, but I am not certain what protections there will be on the map copy object.

10.6.4 Return Values

Table 50 describes the values returned at the completion of the request and the conditions under which each value is returned.

$return!$	C1	C2	C3
$Kern\_success$	T	T	T
$Kern\_protection\_failure$	T	T	F
$Kern\_invalid\_address$	T	F	-
$Kern\_invalid\_argument$	F	-	-

Table 50: Return Values for **vm_write**

$RV\,VmWriteSuccessful$
$VmWriteOutputs$
$C1\,VmWritePageAligned$
$C2\,VmWriteGoodAddress$
$C3\,VmWriteWritable$

$return! = Kern\_success$

$RV\,VmWriteProtectFail$
$VmWriteOutputs$
$C1\,VmWritePageAligned$
$C2\,VmWriteGoodAddress$
$NotC3\,VmWriteWritable$

$return! = Kern\_protection\_failure$

$$\begin{array}{l} \textit{RVVmWriteBadAddress} \\ \hline \textit{VmWriteOutputs} \\ C1\,\textit{VmWritePageAligned} \\ \textit{NotC2}\,\textit{VmWriteGoodAddress} \\ \hline \textit{return}! = \textit{Kern\_invalid\_address} \end{array}$$

$$\begin{array}{l} \textit{RVVmWriteInvalidArg} \\ \hline \textit{VmWriteOutputs} \\ \textit{NotC1}\,\textit{VmWritePageAligned} \\ \hline \textit{return}! = \textit{Kern\_invalid\_argument} \end{array}$$

Review Note:
The prototype also checks (after C1) whether the size of the $MapCopy$ is zero. If so, it returns $Kern\_success$ without checking conditions C2 and C3. No changes are made to the state. Since, when the size is zero, conditions C2 and C3 are automatically true and no words are changed below, this circumstance is covered by the $Kern\_success$ case, and we do not state the extra criterion.

10.6.5 State Changes

A successful **vm_write** request writes the data to the memory pages associated with the specified area of virtual memory. The data written into the address space of the target task originated from the address space of some task (e.g., it was read from that address space using **vm_read**).

$$\begin{array}{l} \textit{VmWriteState} \\ \hline \Delta\ \textit{VAWord} \\ \textit{target\_task}? : \textit{TASK} \\ \textit{address}? : \textit{VIRTUAL\_ADDRESS} \\ \textit{data}? : \textit{MapCopy} \\ \hline \forall\, x : 0\,..\,\textit{data}?.\textit{size} - 1 \\ \mid (\textit{address}?, x) \in \operatorname{dom} \textit{Relative\_addr} \\ \bullet\ \textit{va\_word}'\,(\textit{target\_task}?, \textit{Relative\_addr}(\textit{address}?, x)) \\ \qquad = \textit{va\_word}\,(\textit{data}?.\textit{task}, \textit{Relative\_addr}(\textit{data}?.\textit{offset}, x)) \end{array}$$

Review Note:
It would be better to model memory maps explicitly (independent of tasks) in the state description instead of just associating virtual addresses with tasks. This would allow map copies to be modeled as a special memory map that has no directly associated task.

10.6.6 Complete Request

The general form of a **vm_write** request received through a task port has the following form.

$$\begin{array}{l} \textit{ProcessingVmWrite} \\ \hline \textit{MessageToVMParameters} \\ \hline \textit{operation}? = \textit{Vm\_write\_id} \end{array}$$

A successful request makes the state changes described in the previous section and creates a kernel reply.

$$VmWriteGood \mathrel{\widehat=} (RVVmWriteSuccessful \land VmWriteState) \\ \gg RequestReturnOnlyStatus$$

An unsuccessful request returns an error status.

$$VmWriteBad \\ \mathrel{\widehat=} (RVVmWriteInvalidArg \lor RVVmWriteBadAddress \lor RVVmWriteProtectFail) \\ \gg RequestNoOp$$

Execution of the request consists of a good execution or an error execution.

$$ExecuteVmWrite \mathrel{\widehat=} VmWriteGood \lor VmWriteBad$$

The full specification for kernel processing of a validated **vm_write** request consists of processing the request followed by its execution.

$$VmWrite \mathrel{\widehat=} ProcessingVmWrite \fatsemi ExecuteVmWrite$$

Review Note:
No interaction with memory managers for the region being written is specified. Any pages not backed by Null_memory must not be locked against writing, but we only have locking information for cached segments of the memory objects.

Section *11*
Notes

11.1 Acronyms

CCA Covert Channel Analysis

CMU Carnegie Mellon University

DTOS Distributed Trusted Operating System

FSPM Formal Security Policy Model

IPC Interprocess Communication

KID Kernel Interface Document

MLS Multi-Level Secure

OSC Object Security Context

OSF Open Software Foundation

OSI Object Security Identifier

SID Security Identifier

SSC Subject Security Context

SSI Subject Security Identifier

VM Virtual Memory

11.2 Glossary

dirty page A page in kernel memory is dirty if the pager associated with the page has not yet been made aware of modifications that have been made to the page.

permission A permission is an access mode enforced by the kernel. The kernel ensures that a service is provided only when the client of the service has the appropriate permission.

precious page A page in kernel memory is precious if the pager associated with the page has indicated that it is not maintaining a copy of the page. Regardless of whether the page is dirty, the kernel must send the contents of the page to the pager before removing the page from memory.

security server A security server is a user space task that provides access computations to the kernel.

Appendix *A*
Bibliography

[1] William R. Bevier and Lawrence M. Smith. A Mathematical Description of the Mach Kernel: Virtual Memory Services (Draft). Technical report, Computational Logic, Incorporated, August 1993.

[2] William R. Bevier and Lawrence M. Smith. A Mathematical Model of the Mach Kernel: Entities and Relations (Draft). Technical report, Computational Logic, Incorporated, April 1993.

[3] William R. Bevier and Lawrence M. Smith. A Mathematical Model of the Mach Kernel: Port Services (Draft). Technical report, Computational Logic, Incorporated, August 1993.

[4] William R. Bevier and Lawrence M. Smith. A Mathematical Model of the Mach Kernel: Task and Thread Services (Draft). Technical report, Computational Logic, Incorporated, August 1993.

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

[6] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[7] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, final draft edition, May 1993.

[8] Secure Computing Corporation. DTOS Kernel Interfaces Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1995. DTOS CDRL A003.

[9] Secure Computing Corporation. DTOS Formal Security Policy Model (FSPM). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996. DTOS CDRL A004.

[10] Secure Computing Corporation. DTOS Lessons Learned Report. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, October 1996. DTOS CDRL A008.

Appendix B
Z Extensions

This section describes "extensions" to the Z specification language that are used in the DTOS FTLS. All of these extensions are defined in terms of constructs in the Z specification language, so they are not technically extensions to the language.

B.1 Disjointness and Partitions

It is often necessary to indicate that each element of a collection of values is unique. For example, consider specifying that $val_1, \ldots, val_n$ are unique values. Since n might be relatively large, it is undesirable to enumerate each pair:

$$val_1 \neq val_2 \land val_1 \neq val_3 \land val_1 \neq val_4 \ldots$$

Although disjoint is part of the Z mathematical toolkit, it addresses disjointness of sets instead of disjointness of values. While we could convert values to singleton sets of values as follows:

$$\mathsf{disjoint}\ \langle \{ val_1 \}, \ldots, \{ val_n \} \rangle$$

this is somewhat inconvenient. Another possibility would be to specify that:

$$\langle val_1, \ldots, val_n \rangle$$

is, when viewed as a function, injective. However, the expression:

$$\langle val_1, \ldots, val_n \rangle \in \mathbb{N} \rightarrowtail\!\!\!\!\!\!\;\mapsto X$$

is a rather unintuitive way to express disjointness.

Instead, the generic predicate $Values\_disjoint$ is defined to state such disjointness properties. The expression $Values\_disjoint \langle val_1, \ldots, val_n \rangle$ denotes that $val_1, \ldots, val_n$ are unique values.

Mach Definition 109

$$
\begin{array}{l}
[X] \\
\hline
Values\_disjoint\ \_ : \mathbb{P}(\mathrm{seq}\ X) \\
\hline
\forall\ val\_seq : \mathrm{seq}\ X \\
\bullet\ Values\_disjoint\ val\_seq \\
\Leftrightarrow (\forall\ i_1, i_2 : \mathbb{N} \mid i_1 \in \mathrm{dom}\ val\_seq \land i_2 \in \mathrm{dom}\ val\_seq \land i_1 \neq i_2 \\
\qquad \bullet\ val\_seq(i_1) \neq val\_seq(i_2))
\end{array}
$$

Similarly, the expression $\langle val_1, \ldots, val_n \rangle\ Values\_partition\ S$ denotes that the values $val_1, \ldots, val_n$ are unique values that together comprise the set val_set.

Mach Definition 110

$$
\begin{array}{l}
[X] \\
\_ \mathit{Values\_partition} \_ : (\mathrm{seq}\ X) \leftrightarrow \mathbb{P}\ X \\
\hline
\forall\ \mathit{val\_seq} : \mathrm{seq}\ X;\ \mathit{val\_set} : \mathbb{P}\ X \\
\bullet\ \mathit{val\_seq}\ \mathit{Values\_partition}\ \mathit{val\_set} \\
\Leftrightarrow (\mathit{Values\_disjoint}\ \mathit{val\_seq} \wedge \mathit{val\_set} = \mathrm{ran}\ \mathit{val\_seq})
\end{array}
$$

B.2 Partial Orders

A partial ordering is a relation that is *reflexive*, *antisymmetric*, and *transitive*.

A reflexive relation is one that relates each element to itself; in other words, the identity relation is contained in every reflexive relation.

An antisymmetric relation is a relation containing no cycles of the form $(val_1, val_2) \in R \wedge (val_2, val_1) \in R$ for distinct val_1 and val_2. Since $(val_2, val_1) \in R$ is equivalent to $(val_1, val_2) \in R^\sim$, a relation is antisymmetric exactly when $(val_1, val_2) \in R \wedge (val_1, val_2) \in R^\sim$ only holds for $val_1 = val_2$. In other words, a relation is antisymmetric when its intersection with its inverse is contained in id.

A relation is transitive when:

$$(val_1, val_2) \in R \wedge (val_2, val_3) \in R \Rightarrow (val_1, val_3) \in R$$

In other words, whenever it is possible to get from val_1 to val_3 through repeated iteration of R, R relates val_1 to val_3 directly. This is equivalent to R^2 being contained in R. For each type X, the following sets of relations are defined:

- $\mathit{Reflexive}[X]$ — the set of all reflexive relations on X
- $\mathit{Anti\_symmetric}[X]$ — the set of all antisymmetric relations on X
- $\mathit{Transitive}[X]$ — the set of all transitive relations on X
- $\mathit{Poset}[X]$ — the set of all relations on X that are posets; this is simply the intersection of $\mathit{Reflexive}[X]$, $\mathit{Anti\_symmetric}[X]$, and $\mathit{Transitive}[X]$

Mach Definition 111

$$
\begin{array}{l}
[X] \\
\mathit{Poset} : \mathbb{P}(X \leftrightarrow X) \\
\mathit{Reflexive} : \mathbb{P}(X \leftrightarrow X) \\
\mathit{Anti\_symmetric} : \mathbb{P}(X \leftrightarrow X) \\
\mathit{Transitive} : \mathbb{P}(X \leftrightarrow X) \\
\hline
\mathit{Poset} = \mathit{Reflexive} \cap \mathit{Anti\_symmetric} \cap \mathit{Transitive} \\
\mathit{Reflexive} = \{R : X \leftrightarrow X \mid \mathrm{id}\ X \subseteq R\} \\
\mathit{Anti\_symmetric} = \{R : X \leftrightarrow X \mid R \cap R^\sim \subseteq \mathrm{id}\ X\} \\
\mathit{Transitive} = \{R : X \leftrightarrow X \mid R^2 \subseteq R\}
\end{array}
$$

B.3 Sequences

The expression $val\_seq\ Add\_value\ val$ is used to denote the sequence resulting from adding the element val to the end of the sequence val_seq. The expression $s\ Wrap\_value\ val$ is used to denote the sequence resulting from replacing the first element of val_seq with val.

Mach Definition 112

$$[X]$$

$$\_\ Add\_value\ \_ : (\text{seq}\ X) \times X \longrightarrow (\text{seq}\ X)$$
$$\_\ Wrap\_value\ \_ : (\text{seq}\ X) \times X \longrightarrow (\text{seq}\ X)$$

$$\forall\ val\_seq : \text{seq}\ X;\ val : X$$
$$\bullet\ val\_seq\ Add\_value\ val = val\_seq \frown \{1 \mapsto val\}$$
$$\wedge\ (\#val\_seq > 0 \Rightarrow val\_seq\ Wrap\_value\ val = val\_seq \oplus \{1 \mapsto val\})$$

The expression $Seq\_plus(S)$ where S is a sequence of numbers returns the sum of the numbers in S.

Mach Definition 113

$$Seq\_plus : \text{seq}\ \mathbb{Z} \longrightarrow \mathbb{Z}$$

$$Seq\_plus(\langle\rangle) = 0$$
$$\forall\ S : \text{seq}_1\ \mathbb{Z}$$
$$\bullet\ Seq\_plus(S) = head(S) + Seq\_plus(tail(S))$$

Appendix C IPC

C.1 IPC Requests

This section describes the **mach_msg** request.

Review Note:
This section has not yet been updated for DTOS. Currently, this section is a direct copy of the corresponding DTMach section with minor changes required for DTOS sections that depend on this section.

C.1.1 Constants and Types

We use the following type to denote **mach_msg** return codes:

$$[MACH\_MSG\_RETURN]$$

The return values defined in Mach are:

$$
\begin{array}{l}
Mm\_no\_op : MACH\_MSG\_RETURN \\
Mm\_send\_msg\_too\_small : MACH\_MSG\_RETURN \\
Mm\_send\_no\_buffer : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_header : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_dest : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_reply : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_notify : MACH\_MSG\_RETURN \\
Mm\_rcv\_invalid\_notify : MACH\_MSG\_RETURN \\
Mm\_rcv\_invalid\_name : MACH\_MSG\_RETURN \\
Mm\_rcv\_in\_set : MACH\_MSG\_RETURN \\
Mm\_rcv\_timed\_out : MACH\_MSG\_RETURN \\
Mm\_rcv\_too\_large : MACH\_MSG\_RETURN \\
Mm\_send\_will\_notify : MACH\_MSG\_RETURN \\
Mm\_success : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_right : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_memory : MACH\_MSG\_RETURN \\
Mm\_send\_invalid\_type : MACH\_MSG\_RETURN \\
Mm\_rcv\_port\_died : MACH\_MSG\_RETURN \\
Mm\_rcv\_port\_changed : MACH\_MSG\_RETURN \\
\hline
Values\_disjoint\langle Mm\_no\_op, Mm\_send\_msg\_too\_small, Mm\_send\_invalid\_header, \\
\quad Mm\_send\_invalid\_dest, Mm\_send\_invalid\_reply, Mm\_send\_invalid\_notify, \\
\quad Mm\_rcv\_invalid\_name, Mm\_rcv\_in\_set, Mm\_rcv\_timed\_out, \\
\quad Mm\_rcv\_too\_large, Mm\_send\_will\_notify, Mm\_success, \\
\quad Mm\_send\_invalid\_right, Mm\_send\_invalid\_memory, Mm\_send\_invalid\_type, \\
\quad Mm\_rcv\_port\_died, Mm\_rcv\_port\_changed\rangle
\end{array}
$$

C.2 mach_msg

Review Note:
This section has not yet been updated for DTOS. Currently, this section is a direct copy of the corresponding DTMach section with minor changes required for DTOS sections that depend on this section.

The request **mach_msg** allows a thread to send and receive messages.[14]

The request has the following input parameters:

- client? — the thread sending or receiving a message
- $msgh$? — the message header; note that this is only relevant when a message is being sent
- $option$? — message options
- $send\_size$? — specifies the size of $msgh$? when a message is being sent
- rcv_size? — specifies the size of $msgh$? when a message is being received
- rcv_name? — specifies the port or port set from which to receive a message when a message is being received
- $time\_out$? — specifies the amount of time to wait for the operation to complete before giving up
- $notify$? — specifies the notification port to use in the case in which notifications are requested
- msg_body? — the message body; note that this is only relevant when a message is being sent

The request has the following output parameters:

- $msgh$! — the message buffer; note that this is only relevant when a message is being received
- rcv_size! — specifies the size of the message when an attempt is made to receive a message that is too large
- msg_body! — the in-line data portion of the message; note that this is only an output in the case when a message is being received
- msg_return! — the status of the request

The request is initiated by a schema of the following form.

[14] The specification of this request is incomplete. See Section C.2.3 for a description of the work that remains to be done.

$$
\begin{array}{|l}
\hline
\textit{MachMsgSignature} \\
\Delta\ DtosExec \\
client? : THREAD \\
msgh : MachMsgHeader \\
option? : \mathbb{P}\ MACH\_MSG\_OPTION \\
send\_size? : \mathbb{N} \\
rcv\_size? : \mathbb{N} \\
rcv\_size! : \mathbb{N} \\
rcv\_name? : NAME \\
time\_out? : \mathbb{N} \\
notify? : NAME \\
msg\_body : MESSAGE\_BODY \\
msg\_return! : MACH\_MSG\_RETURN \\
\hline
client? \in \mathrm{dom}\ owning\_task \\
owning\_task\ client? \in \underline{t}ask\_exists \\
\hline
\end{array}
$$

where the meaning of the parameters is as described earlier.[15]

If $option?$ includes neither $Mach\_send\_msg$ nor $Mach\_rcv\_msg$, then no processing occurs.[16]

$$
\begin{array}{|l}
\hline
\textit{MachMsgNoOp} \\
\Xi\ DtosExec \\
MachMsgSignature \\
\hline
\{\ Mach\_send\_msg, Mach\_rcv\_msg\ \} \cap option? = \varnothing \\
msg\_return! = Mm\_no\_op \\
\hline
\end{array}
$$

C.2.1 Message Send

The **mach_msg** request can be used to send a message by including $Mach\_send\_msg$ in $option?$ and not including $Mach\_rcv\_msg$.

$$
\begin{array}{|l}
\hline
\textit{MachMsgSend} \\
MachMsgSignature \\
\hline
Mach\_send\_msg \in option? \\
Mach\_rcv\_msg \notin option? \\
\hline
\end{array}
$$

There are four general cases to consider:

- An error condition occurs during the initial processing, and the request is a no-op.
- A subsequent error condition occurs and the message is returned through a pseudo-receive operation.
- A subsequent error condition occurs and some of the message is lost during delivery.

[15] Note that the $msgh?$ and $msgh!$ parameters are both represented by $msgh$. Similarly, $msg\_body?$ and $msg\_body!$ are both represented by msg_body.

[16] The DTMach Kernel Interface does not define a return status for this case. We have introduced the return status Mm_no_op to denote the return status in this case.

- The message is successfully delivered.

The first case is discussed in Section C.2.1.1. The remaining cases are described in Section C.2.1.2.

C.2.1.1 Initial Processing We use the following schema to describe send operations that are processed as no-ops due to error conditions that arise during the initial processing of the request:

```
┌─ MachMsgSendNoOp ──────────────────────────
│ Ξ Dtos
│ MachMsgSend
└────────────────────────────────────────────
```

If $send\_size?$ is too small, then an error message is returned and no further processing occurs. We define the following constant to denote the minimum send size.

$$Min\_send\_size : \mathbb{N}$$

The case in which the message is too small is specified as follows:

```
┌─ MachMsgSendMsgTooSmall ───────────────────
│ MachMsgSendNoOp
├──────────
│ send_size? < Min_send_size
│ msg_return! = Mm_send_msg_too_small
└────────────────────────────────────────────
```

If there is not enough memory available for the kernel to process the request, then an error message is returned and no further processing occurs. We use the following predicate to indicate when there is insufficient memory available:

$$cannot\_allocate\_send\_buffer\ \_ : \mathbb{P}\ Mach$$

The specification of the processing is as follows:[17]

```
┌─ MachMsgSendSizeOk ────────────────────────
│ send_size? : ℕ
├──────────
│ send_size? ≥ Min_send_size
└────────────────────────────────────────────
```

```
┌─ MachMsgSendNoBuffer ──────────────────────
│ MachMsgSendNoOp
│ MachMsgSendSizeOk
├──────────
│ cannot_allocate_send_buffer(θMach)
│ msg_return! = Mm_send_no_buffer
└────────────────────────────────────────────
```

[17] For convenience, we define a schema representing the negation of the earlier tests before defining the schema representing the processing for a given case. For example, the schema $MachMsgSendSizeOk$ is the negation of the previously described test of whether $send\_size?$ is too small. The schema $MachMsgSendNoBuffer$ uses $MachMsgSendSizeOk$ to define the processing for the case in which there is insufficient memory available to process the request.

If the rights specified for the local or remote port are invalid, then an error message is returned and no further processing occurs. The rights specified for the remote port are valid only if they provide the receiver with either a $Send$ or $Send\_once$ right. Thus, the rights are invalid if they do not include any of Mmt_make_send, Mmt_copy_send, Mmt_move_send, $Mmt\_make\_send\_once$, and $Mmt\_move\_send\_once$. We define the set $TRANSFER\_SEND\_RIGHTS$ to denote this set of values of type $MACH\_MSG\_TYPE$.

$$TRANSFER\_SEND\_RIGHTS == \{ Mmt\_make\_send, Mmt\_copy\_send, Mmt\_move\_send, \\ \quad Mmt\_make\_send\_once, Mmt\_move\_send\_once \}$$

The remote port rights are valid exactly when they contain an element of this set. Similarly, the local port rights are valid when they contain an element of $TRANSFER\_SEND\_RIGHTS$. In addition, the local port rights are also valid when they are empty and the local port is null.

The specification for the case in which either the remote or local port rights are invalid is as follows:

$$\begin{array}{|l}
\hline
MachMsgSendCanAllocateBuffer \\
\hline
Mach \\
MachMsgSendSizeOk \\
\hline
\neg\ cannot_allocate_send_buffer(\theta Mach) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
MachMsgSendInvalidHeader \\
\hline
MachMsgSendCanAllocateBuffer \\
MachMsgSendNoOp \\
\hline
(msgh.remote_rights \notin TRANSFER_SEND_RIGHTS \lor \\
\quad TRANSFER_SEND_RIGHTS \cap msgh.local_rights = \varnothing \land \\
\quad\quad (msgh.local_rights \neq \varnothing \lor \\
\quad\quad msgh.local_port \neq Mach_port_null) \lor \\
\quad \neg\ msgh.remote_rights \in Recognized_transfer_options \lor \\
\quad \neg\ msgh.local_rights \subseteq Recognized_transfer_options) \\
msg_return! = Mm_send_invalid_header \\
\hline
\end{array}$$

Otherwise, if the client task does not have the right required by $msgh.remote\_rights$, then an error message is returned and no further processing occurs. We use the following function to denote the right required for each type of transfer:

$$\begin{array}{|l}
Required_right : Recognized_transfer_options \longrightarrow RIGHT \\
\hline
Required_right = \\
\quad \{ Mmt_make_send \mapsto Receive, \\
\quad Mmt_move_send \mapsto Send, \\
\quad Mmt_copy_send \mapsto Send, \\
\quad Mmt_make_send_once \mapsto Receive, \\
\quad Mmt_move_send_once \mapsto Send_once, \\
\quad Mmt_move_receive \mapsto Receive \}
\end{array}$$

This function captures the following semantics of port right transfers in Mach:

- A receive right can be moved or used to create a send or send-once right.

- A send right can be moved or copied.
- A send-once right can be moved.

Using this function, we specify the case in which the destination port is valid as follows:

$$
\begin{array}{l}
\underline{\quad MachMsgSendValidHeader \quad}\\
MachMsgSendCanAllocateBuffer\\
msgh : MachMsgHeader\\
\hline
msgh.remote\_rights \in TRANSFER\_SEND\_RIGHTS\\
(TRANSFER\_SEND\_RIGHTS \cap msgh.local\_rights \neq \varnothing \lor\\
\quad (msgh.local\_rights = \varnothing \land\\
\quad msgh.local\_port = Mach\_port\_null))\\
msgh.remote\_rights \in Recognized\_transfer\_options\\
msgh.local\_rights \subseteq Recognized\_transfer\_options
\end{array}
$$

$$
\begin{array}{l}
\underline{\quad MachMsgSendInvalidDest \quad}\\
MachMsgSendValidHeader\\
MachMsgSendNoOp\\
\hline
(\textbf{let}\ needed\_rights == Required\_right(msgh.remote\_rights);\\
port == named\_port(owning\_task(client?), msgh.remote\_port) \bullet\\
\quad \forall i : \mathbb{N} \bullet\\
\quad\quad (owning\_task(client?), port, msgh.remote\_port, needed\_rights, i)\\
\quad\quad\quad \notin \underline{p}ort\_right\_rel)\\
msg\_return! = Mm\_send\_invalid\_dest
\end{array}
$$

Otherwise, if the client task does not have the right specified in $msgh.local\_rights$, then an error message is returned and no further processing occurs. Note that if the client task specifies Mmt_move_send or $Mmt\_move\_send\_once$ in $msgh.remote\_rights$, then it loses a reference to $msgh.remote\_port$. This change in the number of references must be accounted for when determining whether the client has sufficient rights for $msgh.local\_port$.

Before defining the functions for manipulating the port name space, we first define the following schema to denote that the previous checks were successful:

$$
\begin{array}{l}
\underline{\quad MachMsgSendValidDest \quad}\\
MachMsgSendValidHeader\\
client? : THREAD\\
\hline
(\textbf{let}\ needed\_rights == Required\_right(msgh.remote\_rights);\\
port == named\_port(owning\_task(client?), msgh.remote\_port) \bullet\\
\quad \exists i : \mathbb{N} \bullet\\
\quad\quad (owning\_task(client?), port, msgh.remote\_port, needed\_rights, i)\\
\quad\quad\quad \in \underline{p}ort\_right\_rel)
\end{array}
$$

We use the following type to denote the kernel data structure defining the port name spaces for each task. This structure has the same format and meaning as the relation $\underline{p}ort\_right\_rel$ in the definition of the Mach system state.

$$PORT\_NAME\_SPACE == \mathbb{P}(TASK \times PORT \times NAME \times RIGHT \times \mathbb{N}_1)$$

The function $Change\_ref\_count$ is used to change the reference count associated with a name and a right in a task's port name space by a specified amount. If subtracting the specified amount from the current count results in a positive value, then the new count is that positive value. Otherwise, the name and right are not present in the port name space returned by this function.

$$Change\_ref\_count : \mathbb{Z} \times TASK \times NAME \times RIGHT \times PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE$$

$$\begin{array}{l} \forall\, task, task_1 : TASK; port : PORT; name, name_1 : NAME; right, right_1 : RIGHT; \\ i : \mathbb{N}_1; n : \mathbb{Z}; pns : PORT\_NAME\_SPACE \bullet \\ \quad (task, port, name, right, i) \in \\ \quad\quad Change\_ref\_count(n, task_1, name_1, right_1, pns) \Leftrightarrow \\ \quad ((task, port, name, right, i) \in pns \wedge \\ \quad\quad (task, name, right) \neq (task_1, name_1, right_1)) \vee \\ \quad ((task, port, name, right, i + n) \in pns \wedge \\ \quad\quad (task, name, right) = (task_1, name_1, right_1)) \end{array}$$

The functions $Change\_receive\_count$, $Change\_send\_count$, and $Change\_send\_once\_count$ use $Change\_ref\_count$ to change the count associated with a receive, send, or send-once right.

$$Change\_receive\_count : \mathbb{Z} \times TASK \times NAME \times PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE$$

$$\begin{array}{l} \forall\, n : \mathbb{Z}; task : TASK; name : NAME; pns : PORT\_NAME\_SPACE \bullet \\ \quad Change\_receive\_count(n, task, name, pns) = \\ \quad\quad Change\_ref\_count(n, task, name, Receive, pns) \end{array}$$

$$Change\_send\_count : \mathbb{Z} \times TASK \times NAME \times PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE$$

$$\begin{array}{l} \forall\, n : \mathbb{Z}; task : TASK; name : NAME; pns : PORT\_NAME\_SPACE \bullet \\ \quad Change\_send\_count(n, task, name, pns) = \\ \quad\quad Change\_ref\_count(n, task, name, Send, pns) \end{array}$$

$$Change\_send\_once\_count : \mathbb{Z} \times TASK \times NAME \times PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE$$

$$\begin{array}{l} \forall\, n : \mathbb{Z}; task : TASK; name : NAME; pns : PORT\_NAME\_SPACE \bullet \\ \quad Change\_send\_once\_count(n, task, name, pns) = \\ \quad\quad Change\_ref\_count(n, task, name, Send\_once, pns) \end{array}$$

The function $Process\_right$ computes a new port name space from an old port name space, a task, a name, and a set of transfer options. If none of the transfer options requires moving a right, then the resulting name space is the same as the input name space. Otherwise, the count for each type of right that is moved is decremented.

$$
\begin{array}{l}
Process\_right : \mathbb{P}\, Recognized\_transfer\_options \times TASK \times NAME \times \\
\quad PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE \\
\hline
\forall\, mmt\_set : \mathbb{P}\, Recognized\_transfer\_options;\ task : TASK;\ name : NAME; \\
pns : PORT\_NAME\_SPACE \bullet \\
\quad (\{ Mmt\_move\_receive, Mmt\_move\_send, Mmt\_move\_send\_once \} \cap mmt\_set = \varnothing \Rightarrow \\
\qquad Process\_right(mmt\_set, task, name, pns) = pns) \wedge \\
\quad (Mmt\_move\_receive \in mmt\_set \Rightarrow \\
\qquad (\mathbf{let}\ mmt\_set_1 == mmt\_set \setminus \{ Mmt\_move\_receive \}; \\
\qquad pns_1 == Change\_receive\_count(1, task, name, pns) \bullet \\
\qquad\quad Process\_right(mmt\_set, task, name, pns) = \\
\qquad\qquad Process\_right(mmt\_set_1, task, name, pns_1))) \wedge \\
\quad (Mmt\_move\_send \in mmt\_set \Rightarrow \\
\qquad (\mathbf{let}\ mmt\_set_1 == mmt\_set \setminus \{ Mmt\_move\_send \}; \\
\qquad pns_1 == Change\_send\_count(1, task, name, pns) \bullet \\
\qquad\quad Process\_right(mmt\_set, task, name, pns) = \\
\qquad\qquad Process\_right(mmt\_set_1, task, name, pns_1))) \wedge \\
\quad (Mmt\_move\_send\_once \in mmt\_set \Rightarrow \\
\qquad (\mathbf{let}\ mmt\_set_1 == mmt\_set \setminus \{ Mmt\_move\_send\_once \}; \\
\qquad pns_1 == Change\_send\_once\_count(1, task, name, pns) \bullet \\
\qquad\quad Process\_right(mmt\_set, task, name, pns) = \\
\qquad\qquad Process\_right(mmt\_set_1, task, name, pns_1)))
\end{array}
$$

Using these functions, the case in which the reply port is invalid can be specified as follows.

$$
\begin{array}{l}
\textbf{— } MachMsgSendInvalidReply \textbf{ —} \\
MachMsgSendValidHeader \\
MachMsgSendNoOp \\
\hline
\mathbf{let}\ needed\_rights == Required\_right (\!| msgh.local\_rights |\!); \\
new\_port\_right\_rel == \\
Process\_right(\{msgh.remote\_rights\}, owning\_task\ client?, \\
\quad msgh.remote\_port, \underline{p}ort\_right\_rel); \\
port == named\_port(owning\_task\ client?, msgh.local\_port) \bullet \\
\quad \exists\, right : RIGHT \mid \\
\qquad right \in needed\_rights \bullet \\
\qquad\quad \forall\, i : \mathbb{N} \bullet \\
\qquad\qquad (owning\_task\ client?, port, msgh.local\_port, right, i) \\
\qquad\qquad\quad \notin new\_port\_right\_rel \\
msg\_return! = Mm\_send\_invalid\_reply
\end{array}
$$

This is analogous to the case in which the destination port is invalid. The main difference is that the reference counts for the destination port are decremented, if necessary, before testing the reply port. When the reply and destination ports are the same, this decrementing can influence whether the reply port is valid.

If the client specifies the $Mach\_send\_cancel$ option and $msgh.notify$ does not denote a receive right, then an error message is returned and no further processing takes place. As before, it is necessary to decrement reference counts associated with earlier right transfers before performing this test.

MachMsgSendValidReply

$$
\begin{array}{l}
MachMsgSendValidHeader \\
MachMsgSendNoOp \\
\hline
\textbf{let } needed\_rights == Required\_right(\!|msgh.local\_rights|\!); \\
new\_port\_right\_rel == \\
Process\_right(\{msgh.remote\_rights\}, owning\_task\ client?, \\
\quad msgh.remote\_port, \underline{p}ort\_right\_rel); \\
port == named\_port(owning\_task\ client?, msgh.local\_port) \bullet \\
\quad \forall\, right : RIGHT \mid \\
\quad\quad right \in needed\_rights \bullet \\
\quad\quad\quad \exists\, i : \mathbb{N} \bullet \\
\quad\quad\quad\quad (owning\_task\ client?, port, msgh.local\_port, right, i) \\
\quad\quad\quad\quad\quad \in new\_port\_right\_rel
\end{array}
$$

MachMsgSendInvalidNotify

$$
\begin{array}{l}
MachMsgSendValidReply \\
MachMsgSendNoOp \\
\hline
Mach\_send\_cancel \in option? \\
\textbf{let } new\_port\_right\_rel == \\
Process\_right(\{msgh.remote\_rights\}, owning\_task\ client?, \\
\quad\quad msgh.remote\_port, \underline{p}ort\_right\_rel) \\
\quad \bullet\ \textbf{let } new\_port\_right\_rel_1 == \\
Process\_right(msgh.local\_rights, owning\_task\ client?, \\
\quad\quad msgh.local\_port, new\_port\_right\_rel) \bullet \\
\quad (\forall\, port : PORT; i : \mathbb{N} \bullet \\
\quad\quad (owning\_task\ client?, port, notify?, Receive, i) \notin new\_port\_right\_rel_1) \\
msg\_return! = Mm\_send\_invalid\_notify
\end{array}
$$

The following schema denotes the case in which the kernel can continue processing the message.

MachMsgSendValid

$$
\begin{array}{l}
MachMsgSignature \\
MachMsgSendValidReply \\
option? : \mathbb{P}\ MACH\_MSG\_OPTION \\
notify? : NAME \\
\hline
Mach\_send\_cancel \notin option? \vee \\
(\textbf{let } new\_port\_right\_rel == \\
Process\_right(\{msgh.remote\_rights\}, owning\_task\ client?, \\
\quad\quad msgh.remote\_port, \underline{p}ort\_right\_rel) \\
\quad \bullet\ \textbf{let } new\_port\_right\_rel_1 == \\
Process\_right(msgh.local\_rights, owning\_task\ client?, \\
\quad\quad msgh.local\_port, new\_port\_right\_rel) \bullet \\
\quad (\exists\, port : PORT; i : \mathbb{N} \bullet \\
\quad\quad (owning\_task\ client?, port, notify?, Receive, i) \in new\_port\_right\_rel_1))
\end{array}
$$

In this case, the message specified by the client is added to the set of messages in user space. Before describing this processing, we first describe functions that convert a message from its format in user space to its format in kernel space.

The function $Msgh\_to\_internal\_msgh$ converts the message header. The $remote\_port$ and $local\_port$ fields are filled in as specified by the two port parameters to the function. The remaining fields are copied without change.

$$Msgh\_to\_internal\_msgh : PORT \times \mathbb{P}\ PORT \times MachMsgHeader \longrightarrow MachInternalHeader$$

$$\begin{array}{l} \forall\ msgh : MachMsgHeader;\ int\_msgh : MachInternalHeader;\ port_1 : PORT;\ port_2 : \mathbb{P}\ PORT \mid \\ \quad Msgh\_to\_internal\_msgh(port_1, port_2, msgh) = int\_msgh \bullet \\ \quad\quad msgh.local\_rights = int\_msgh.local\_rights \wedge \\ \quad\quad msgh.remote\_rights = int\_msgh.remote\_rights \wedge \\ \quad\quad msgh.size = int\_msgh.size \wedge \\ \quad\quad msgh.operation = int\_msgh.operation \wedge \\ \quad\quad port_1 = int\_msgh.remote\_port \wedge \\ \quad\quad port_2 = int\_msgh.local\_port \end{array}$$

Editorial Note:
The previous definition used to state that the $complex$ field of the internal message header was copied from the user space message header. It appears that it is really generated by the kernel parsing the message.

The function $Msg\_data\_to\_msg\_value$ converts an element of type MSG_DATA to an element of type MSG_VALUE.

$$Msg\_data\_to\_msg\_value : MSG\_DATA \longrightarrow MSG\_VALUE$$

$$\begin{array}{l} \forall\ msg\_data : MSG\_DATA \bullet \\ \quad Msg\_data\_to\_msg\_value\ msg\_data = V\_data(msg\_data, V\_data\_in) \end{array}$$

The function $Msg\_data\_seq\_to\_msg\_value\_seq$ converts an element of type $\mathrm{seq}\ MSG\_DATA$ to an element of type $\mathrm{seq}\ MSG\_VALUE$.

$$Msg\_data\_seq\_to\_msg\_value\_seq : \mathrm{seq}\ MSG\_DATA \longrightarrow \mathrm{seq}\ MSG\_VALUE$$

$$\begin{array}{l} \forall\ data\_seq : \mathrm{seq}\ MSG\_DATA \bullet \\ \quad Msg\_data\_seq\_to\_msg\_value\_seq\ data\_seq = Msg\_data\_to\_msg\_value \circ data\_seq \end{array}$$

The function $Msge\_to\_internal\_msge$ converts a single element of a message body. Elements in a message in user space are either In_line or Out_of_line. The former are converted to Msg_value entries, and the latter are converted to Msg_region entries. The function's task parameter is associated with the element to record the task in whose space out-of-line data and port rights should later be resolved.

$$Msge\_to\_internal\_msge : TASK \times Msg\_element \longrightarrow Internal\_element$$

$$\begin{array}{l} \forall\ n : \mathbb{N};\ mach\_msg\_type : MACH\_MSG\_TYPE;\ data\_seq : \mathrm{seq}\ MSG\_DATA; \\ va : VIRTUAL\_ADDRESS;\ int\_msge : Internal\_element;\ task : TASK;\ olsd : OLSD \bullet \\ \quad (\mathbf{let}\ value\_seq == Msg\_data\_seq\_to\_msg\_value\_seq\ data\_seq \bullet \\ \quad\quad (Msge\_to\_internal\_msge(task, In\_line(n, mach\_msg\_type, data\_seq)) \\ \quad\quad\quad\quad = int\_msge \\ \quad\quad\quad \Rightarrow int\_msge = Msg\_value(n, mach\_msg\_type, (task, value\_seq))) \\ \quad\quad \wedge\ (Msge\_to\_internal\_msge(task, Out\_of\_line(n, mach\_msg\_type, va, olsd)) \\ \quad\quad\quad\quad = int\_msge \\ \quad\quad\quad \Rightarrow int\_msge = Msg\_region(n, mach\_msg\_type, (task, va, olsd)))) \end{array}$$

The function $Msgb\_to\_internal\_msgb$ converts a message body by applying $Msge\_to\_internal\_msge$ to each element in the body.

$$Msgb\_to\_internal\_msgb : TASK \times MESSAGE\_BODY \longrightarrow INTERNAL\_BODY$$

$$\begin{array}{l}\forall\, msgb : MESSAGE\_BODY;\ int\_msgb : INTERNAL\_BODY;\ task : TASK \mid \\ \quad Msgb\_to\_internal\_msgb(task, msgb) = int\_msgb \bullet \\ \quad\quad \#msgb = \#int\_msgb \land \\ \quad\quad (\forall\, i : \mathbb{N} \mid i \in \operatorname{dom} msgb \bullet \\ \quad\quad\quad int\_msgb(i) = Msge\_to\_internal\_msge(task, msgb(i)))\end{array}$$

Finally, a message in user space is converted to a message in user space by using $Msgh\_to\_internal\_msgh$ to convert the header and using $Msgb\_to\_internal\_msgb$ to convert the body.

$$\begin{array}{l}Msg\_to\_internal\_msg : \mathbb{N} \times \mathbb{N} \times \mathbb{P}\, MACH\_MSG\_OPTION \times PORT \times PORT \times TASK \times \\ \quad Message \longrightarrow InternalMessage\end{array}$$

$$\begin{array}{l}\forall\, task : TASK;\ port_1, port_2 : PORT;\ msg : Message;\ int\_msg : InternalMessage; \\ \quad current\_time, time\_out : \mathbb{N};\ option? : \mathbb{P}\, MACH\_MSG\_OPTION \\ \mid Msg\_to\_internal\_msg(current\_time, time\_out, option?, port_1, port_2, task, msg) \\ \quad = int\_msg \\ \bullet\ Msgh\_to\_internal\_msgh(port_1, \{port_2\}, msg.header) = int\_msg.header \land \\ \quad\quad Msgb\_to\_internal\_msgb(task, msg.body) = int\_msg.body \land \\ \quad int\_msg.option = option? \land \\ \quad int\_msg.time\_out\_at = \mathbf{if}\ Mach\_send\_timeout \in option? \\ \quad\quad \mathbf{then}\ \{\ current\_time + time\_out\ \} \\ \quad\quad \mathbf{else}\ \varnothing \land \\ \quad int\_msg.status = Msg\_stat\_send \land \\ \quad int\_msg.error = \varnothing\end{array}$$

Note that:

- The $time\_out\_at$ field is set to indicate the earliest time at which the send request can time out.

  If the client specified a time out was desired, then this field is set to the current time plus the specified time out duration. Otherwise, the $time\_out\_at$ field is set to $\varnothing$ to denote that the send request should block rather than time out.

- The $status$ field is set to indicate that the message should be processed as part of a send request.

- The $error$ field is initialized to $\varnothing$.

The function $Msgh\_to\_internal\_msgh$ requires inputs indicating the remote and local ports. The remote port can be determined by using $named\_port$ to resolve the remote port name in the task's name space. When the local port name is not null, the same approach can be used to determine the local port. In the cases in which the local port name is null, we use $Null\_port$ to denote the local port.

$$Null\_port : PORT$$

Before the message is moved into kernel space, the appropriate reference counts are decremented and the make send count is incremented for any port for which a send right was made.

The function $Update\_ms\_count$ defines the changes that need to be made to $make\_send\_count$. The count for the remote port must be incremented by 1 if a send right was made. Similarly, the count for the local port must be incremented by 1 if it exists and a send right was made for it.

$$
\begin{array}{l}
Update\_ms\_count : (PORT \times \mathbb{P}\ Recognized\_transfer\_options) \times \\
\quad (PORT \times \mathbb{P}\ Recognized\_transfer\_options) \times \mathbb{P}\ PORT \times \\
\quad\quad (PORT \mathrel{+\!\!\!\rightarrow} \mathbb{N}) \longrightarrow (PORT \mathrel{+\!\!\!\rightarrow} \mathbb{N}) \\
\hline
\forall\, port_1, port_2 : PORT;\ old\_ms\_count : PORT \mathrel{+\!\!\!\rightarrow} \mathbb{N}; \\
mmt\_set_1, mmt\_set_2 : \mathbb{P}\ MACH\_MSG\_TYPE;\ port\_set : \mathbb{P}\ PORT \bullet \\
\quad Update\_ms\_count((port_1, mmt\_set_1), (port_2, mmt\_set_2), port\_set, old\_ms\_count) = \\
\quad \textbf{if}\ port_2 \notin port\_set \\
\quad\quad \textbf{then}\ old\_ms\_count \oplus \{\, port_1 \mapsto \\
\quad\quad\quad old\_ms\_count\ port_1 + \#(\{\, Mmt\_make\_send \,\} \cap mmt\_set_1) \,\} \\
\quad \textbf{else if}\ port_1 = port_2 \\
\quad\quad \textbf{then}\ old\_ms\_count \oplus \{\, port_1 \mapsto \\
\quad\quad\quad old\_ms\_count\ port_1 + \#(\{\, Mmt\_make\_send \,\} \cap mmt\_set_1) + \\
\quad\quad\quad\quad \#(\{\, Mmt\_make\_send \,\} \cap mmt\_set_2) \,\} \\
\quad \textbf{else} \\
\quad\quad old\_ms\_count \oplus \{\, port_1 \mapsto \\
\quad\quad\quad old\_ms\_count\ port_1 + \#(\{\, Mmt\_make\_send \,\} \cap mmt\_set_1), \\
\quad\quad\quad port_2 \mapsto old\_ms\_count\ port_2 + \#(\{\, Mmt\_make\_send \,\} \cap mmt\_set_2) \,\}
\end{array}
$$

Note that the above definition accounts for the possibility that the local and remote ports are the same by counting the send rights made for either port against the common port.

The function $Update\_name\_space$ performs any necessary decrementing of the reference counts for the local and remote ports. It does so by first using $Process\_right$ to address the remote port and then using $Process\_right$ on the result to address the local port.

$$
\begin{array}{l}
Update\_name\_space : (\mathbb{P}\ MACH\_MSG\_TYPE \times NAME) \times \\
\quad (\mathbb{P}\ MACH\_MSG\_TYPE \times NAME) \times \\
TASK \times PORT\_NAME\_SPACE \longrightarrow PORT\_NAME\_SPACE \\
\hline
\forall\, mmt\_set_1, mmt\_set_2 : \mathbb{P}\ MACH\_MSG\_TYPE;\ task : TASK;\ name_1, name_2 : NAME; \\
pns : PORT\_NAME\_SPACE \bullet \\
\quad Update\_name\_space((mmt\_set_1, name_1), (mmt\_set_2, name_2), task, pns) = \\
\quad\quad (\textbf{let}\ pns_1 == Process\_right(mmt\_set_1, task, name_1, pns) \bullet \\
\quad\quad\quad Process\_right(mmt\_set_2, task, name_2, pns_1))
\end{array}
$$

Using the previously defined functions, the entering of a send message request into kernel space can be specified as follows:

MachMsgSendStart

$$
\begin{array}{l}
MachMsgSendValid \\
\hline
\exists\, message : MESSAGE;\ msg : Message; \\
\qquad port_1, port_2 : PORT;\ task : TASK \bullet \\
\quad message \in \underline{m}essage\_exists' \setminus \underline{m}essage\_exists \land \\
\quad msg.header = msgh \land \\
\quad msg.body = msg\_body \land \\
\quad task = owning\_task\ client? \land \\
\quad port_1 = named\_port(task, msgh.remote\_port) \land \\
\quad (port_2 = \mathbf{if}\ (task, msgh.local\_port) \in \mathrm{dom}\, named\_port \\
\qquad \mathbf{then}\ named\_port(task, msgh.local\_port) \\
\qquad \mathbf{else}\ Null\_port) \land \\
\quad \underline{m}sg\_contents' = \underline{m}sg\_contents \cup \{\, message \mapsto \\
\qquad Msg\_to\_internal\_msg(\underline{h}ost\_time, time\_out?, option?, port_1, port_2, \\
\qquad\quad task, msg)\,\} \\
\quad \land\ \underline{m}ake\_send\_count' = \\
\qquad (\mathbf{let}\ pair_1 == (port_1, \{msgh.remote\_rights\}); \\
\qquad pair_2 == (port_2, msgh.local\_rights) \bullet \\
\qquad\quad Update\_ms\_count(pair_1, pair_2, \underline{p}ort\_exists, \underline{m}ake\_send\_count)) \\
(\mathbf{let}\ pair_1 == (\{msgh.remote\_rights\}, msgh.remote\_port); \\
pair_2 == (msgh.local\_rights, msgh.local\_port) \bullet \\
\quad \underline{p}ort\_right\_rel' = Update\_name\_space(pair_1, pair_2, owning\_task\ client?, \\
\qquad \underline{p}ort\_right\_rel))
\end{array}
$$

C.2.1.2 Kernel Processing In this section, we describe the processing of messages in kernel space that are not yet queued at a port.

The function $Unprocessed\_rights$ returns the set of port rights in transit that must be processed before a message can be enqueued. An element $(message, i, j)$ belongs to the resulting set exactly when the i^{th} element of $message$'s body is a data element whose j^{th} entry is an unresolved port right. Note that regardless of the types of the data elements in a message body, no rights are transferred unless the $complex$ field of the message header indicates rights are being transferred.

$$
\begin{array}{l}
Unprocessed\_rights : Mach \longrightarrow \mathbb{P}(MESSAGE \times \mathbb{N} \times \mathbb{N}) \\
\hline
\forall\, mach\_st : Mach \bullet \\
\quad Unprocessed\_rights\ mach\_st = \{message : MESSAGE;\ i, j : \mathbb{N} \mid \\
\qquad message \in mach\_st.\underline{m}essage\_exists \land \\
\qquad Co\_carries\_rights \in (mach\_st.\underline{m}sg\_contents\ message).header.complex \land \\
\qquad (mach\_st.\underline{m}sg\_contents\ message).status = Msg\_stat\_send \land \\
\qquad (\mathbf{let}\ int\_msgb == (mach\_st.\underline{m}sg\_contents\ message).body \bullet \\
\qquad\quad i \in \mathrm{dom}\, int\_msgb \land \\
\qquad\quad (\exists\, n : \mathbb{N};\ mach\_msg\_type : MACH\_MSG\_TYPE; \\
\qquad\quad value\_seq : \mathrm{seq}\, MSG\_VALUE;\ task : TASK; \\
\qquad\quad msg\_data : MSG\_DATA;\ v\_data\_l : V\_DATA\_LOCATION \\
\qquad\qquad \bullet\ int\_msgb(i) = Msg\_value(n, mach\_msg\_type, (task, value\_seq)) \land \\
\qquad\qquad mach\_msg\_type \in Recognized\_transfer\_options \land \\
\qquad\qquad j \in \mathrm{dom}\, value\_seq \land \\
\qquad\qquad value\_seq(j) = V\_data(msg\_data, v\_data\_l)))\}
\end{array}
$$

The function $Unprocessed\_memories$ returns the set of memory objects in transit that must be processed before a message can be enqueued. An element $(message, i)$ belongs to the resulting set exactly when the i^{th} element of $message$'s body is an unprocessed out-of-line data element. Note that regardless of the types of the data elements in a message body, no memories are transferred unless the $complex$ field of the message header indicates memories are being transferred.

$$
\begin{array}{l}
Unprocessed\_memories : Mach \longrightarrow \mathbb{P}(MESSAGE \times \mathbb{N}) \\
\hline
\forall\, mach\_st : Mach \bullet \\
\quad Unprocessed\_memories\; mach\_st = \{ message : MESSAGE;\; i : \mathbb{N} \mid \\
\quad\quad message \in mach\_st.\underline{m}essage\_exists \land \\
\quad\quad Co\_carries\_memory \in (mach\_st.\underline{m}sg\_contents\; message).header.complex \land \\
\quad\quad (mach\_st.\underline{m}sg\_contents\; message).status = Msg\_stat\_send \land \\
\quad\quad (\mathbf{let}\; int\_msgb == (mach\_st.\underline{m}sg\_contents\; message).body \bullet \\
\quad\quad\quad i \in \mathrm{dom}\, int\_msgb \land \\
\quad\quad\quad (\exists\, n : \mathbb{N};\; mach\_msg\_type : MACH\_MSG\_TYPE;\; olsd : OLSD; \\
\quad\quad\quad task : TASK;\; va : VIRTUAL\_ADDRESS \bullet \\
\quad\quad\quad\quad int\_msgb(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)))) \}
\end{array}
$$

The function $Element\_type$ returns the type of an element in a message body.

$$
\begin{array}{l}
Element\_type : Internal\_element \longrightarrow MACH\_MSG\_TYPE \\
\hline
\forall\, inte : Internal\_element;\; n : \mathbb{N};\; mach\_msg\_type : MACH\_MSG\_TYPE; \\
\quad value\_seq : \mathrm{seq}\, MSG\_VALUE;\; task : TASK; \\
\quad va : VIRTUAL\_ADDRESS;\; olsd : OLSD; \\
\quad memory : MEMORY;\; offset : OFFSET \mid \\
\quad\quad inte \in \\
\quad\quad\quad \{ Msg\_value(n, mach\_msg\_type, (task, value\_seq)), \\
\quad\quad\quad Msg\_region(n, mach\_msg\_type, (task, va, olsd)), \\
\quad\quad\quad Transit\_memory(n, mach\_msg\_type, (task, memory, offset)) \} \bullet \\
\quad\quad\quad\quad Element\_type\; inte = mach\_msg\_type
\end{array}
$$

The set $Invalid\_msg\_types$ indicates the set of message elements having invalid data types. An element $(message, i)$ belongs to the resulting set exactly when the type specified for the $i^t h$ element of $message$'s body is invalid. The set $Valid\_data\_types$ defines the set of valid data types.

$$
\begin{array}{l}
Valid\_data\_types : \mathbb{P}\, MACH\_MSG\_TYPE \\
Invalid\_msg\_types : Mach \longrightarrow \mathbb{P}(MESSAGE \times \mathbb{N}) \\
\hline
Recognized\_transfer\_options \subseteq Valid\_data\_types \\
\forall\, mach\_st : Mach \bullet \\
\quad Invalid\_msg\_types\; mach\_st = \{ message : MESSAGE;\; i : \mathbb{N} \mid \\
\quad\quad message \in mach\_st.\underline{m}essage\_exists \land \\
\quad\quad i \in \mathrm{dom}(mach\_st.\underline{m}sg\_contents\; message).body \land \\
\quad\quad Element\_type((mach\_st.\underline{m}sg\_contents\; message).body(i)) \notin Valid\_data\_types \}
\end{array}
$$

The set $Processed\_messages$ indicates the set of messages that are not yet enqueued but require no further processing. In other words, these are messages that have no elements with invalid data types or unprocessed rights or memories and that are not present in any message queue.

$$
\begin{array}{l}
Processed\_messages : Mach \longrightarrow \mathbb{P}\, MESSAGE \\
\hline
\forall\, mach\_st : Mach \bullet \\
\quad Processed\_messages\ mach\_st = \\
\quad\quad mach\_st.\underline{m}essage\_exists \setminus \\
\quad\quad\quad (\{ message : MESSAGE; i, j : \mathbb{N} \mid \\
\quad\quad\quad\quad (message, i, j) \in Unprocessed\_rights\ mach\_st \bullet message \} \cup \\
\quad\quad\quad \{ message : MESSAGE; i : \mathbb{N} \mid \\
\quad\quad\quad\quad (message, i) \in Invalid\_msg\_types\ mach\_st \bullet message \} \cup \\
\quad\quad\quad \{ message : MESSAGE; i : \mathbb{N} \mid \\
\quad\quad\quad\quad (message, i) \in Unprocessed\_memories\ mach\_st \bullet message \} \cup \\
\quad\quad\quad \{ message : MESSAGE \mid (\exists\, port : PORT \bullet \\
\quad\quad\quad\quad message \in \operatorname{ran}(mach\_st.\underline{m}essage\_in\_port\_rel\ port)) \} \cup \\
\quad\quad\quad \{ message : MESSAGE \mid (mach\_st.\underline{m}sg\_contents\ message).status \\
\quad\quad\quad\quad \neq Msg\_stat\_send \})
\end{array}
$$

The function $Address\_to\_index$ is used to convert a virtual address into a page index.

$$
Address\_to\_index : VIRTUAL\_ADDRESS \longrightarrow PAGE\_INDEX
$$

Before describing the processing of message elements, we define the following schema to represent parts of the processing that are common to the various cases to be considered:

$$
\begin{array}{l}
\textit{GeneralSendProcessing} \\
\hline
\Delta\ DtosExec \\
message : MESSAGE \\
i, n : \mathbb{N} \\
int\_msg_1, int\_msg_2 : InternalMessage \\
v\_data\_l : V\_DATA\_LOCATION \\
task : TASK \\
value\_seq_1, value\_seq_2 : \operatorname{seq} MSG\_VALUE \\
mach\_msg\_type : MACH\_MSG\_TYPE \\
va : VIRTUAL\_ADDRESS \\
olsd : OLSD \\
memory : MEMORY \\
offset : OFFSET \\
error : MSG\_ERROR \\
page\_index : PAGE\_INDEX \\
\hline
message \in \underline{m}essage\_exists \\
int\_msg_1 = \underline{m}sg\_contents\ message \\
i \in \operatorname{dom}(int\_msg_1.body) \\
int\_msg_1.body(i) \in \\
\quad \{ Msg\_value(n, mach\_msg\_type, (task, value\_seq_1)), \\
\quad Msg\_region(n, mach\_msg\_type, (task, va, olsd)), \\
\quad Transit\_memory(n, mach\_msg\_type, (task, memory, offset)) \} \\
\underline{m}sg\_contents' = \underline{m}sg\_contents \oplus \{ message \mapsto int\_msg_2 \} \\
int\_msg_2.header = int\_msg_1.header \\
int\_msg_2.option = int\_msg_1.option \\
int\_msg_2.time\_out\_at = int\_msg_1.time\_out\_at \\
int\_msg_1.error \neq \varnothing \Rightarrow int\_msg_2.error = int\_msg_1.error \\
page\_index = Address\_to\_index\ va
\end{array}
$$

This schema requires that $message$ is an existing message and i is a valid index for the body of the message associated with $message$. The processing of the element is accomplished by modifying the body of the message. The components int_msg_1 and int_msg_2 are introduced to denote the initial and final values for the message. It is required that the $header$, $option$, and $time\_out\_at$ fields of the message are not altered. Furthermore, it is required that the $error$ field cannot be altered if it is nonempty. The remaining components of the schema are introduced to define the general form of the i^{th} element of the message body.

The function $replace\_entry$ replaces a specified entry in a sequence with a specified value.

$$
\begin{array}{l}
[X] \\
replace\_entry : \mathbb{N}_1 \times X \times \mathrm{seq}\, X \longrightarrow \mathrm{seq}\, X \\
\hline
\forall\, i : \mathbb{N}_1;\; x : X;\; x\_seq : \mathrm{seq}\, X \mid i \in \mathrm{dom}\, x\_seq \bullet \\
\quad replace\_entry(i, x, x\_seq) = x\_seq \oplus \{\, i \mapsto x \,\}
\end{array}
$$

The function $Data\_to\_name$ converts an element of type MSG_DATA to an element of type $NAME$. It is assumed that this function is an injection.

$$
Data\_to\_name : MSG\_DATA \rightarrowtail\!\!\!\!\!\rightarrow NAME
$$

From the schema $GeneralSendProcessing$, we build the following schema for processing port rights:

$$
\begin{array}{l}
GeneralSendProcessing2 \\
\hline
GeneralSendProcessing \\
j : \mathbb{N} \\
value\_seq_2 : \mathrm{seq}\, MSG\_VALUE \\
port : PORT \\
msg\_data : MSG\_DATA \\
name : NAME \\
\hline
int\_msg_1.body(i) = Msg\_value(n, mach\_msg\_type, (task, value\_seq_1)) \\
mach\_msg\_type \in Recognized\_transfer\_options \\
Co\_carries\_rights \in int\_msg_1.header.complex \\
j \in \mathrm{dom}\, value\_seq_1 \\
value\_seq_1(j) = V\_data(msg\_data, v\_data\_l) \\
name = Data\_to\_name\ msg\_data \\
port = \mathbf{if}\ (task, name) \in \mathrm{dom}\, named\_port \wedge \\
\quad (\exists\, k : \mathbb{N} \bullet \\
\quad (task, named\_port(task, name), name, Required\_right(mach\_msg\_type), k) \in \\
\qquad \underline{p}ort\_right\_rel) \\
\qquad\quad \mathbf{then}\ named\_port(task, name) \\
\qquad\quad \mathbf{else}\ Null\_port \\
value\_seq_2 = replace\_entry(j, V\_port(port, v\_data\_l), value\_seq_1) \\
int\_msg_2.body = \\
\quad replace\_entry(i, Msg\_value(n, mach\_msg\_type, (task, value\_seq_2)), \\
\qquad int\_msg_1.body)
\end{array}
$$

This schema requires that the message element being processing is of the Msg_value form and the type of the message element indicates that a port right is being transferred. The component j indicates the index into $value\_seq_1$ denoting the right to be processed. The component

$value\_seq_2$ is introduced to denote the new sequence of values to be stored as the i^{th} element in the body. The new sequence is obtained by replacing the j^{th} entry of the original sequence with an entry indicating the port to which the transferred right resolves. The component $port$ is introduced to denote this port. The component $name$ is introduced to represent the $name$ of the transferred right. The name is defined by the data in the j^{th} element of the sequence. If the name is in the task's name space and the task has the appropriate rights to transfer the port, then $port$ is defined to be the port associated with the name in the task's name space. Otherwise, $port$ is defined to be the $Null\_port$.

If a message contains an element with an invalid data type, then progress can be made in processing the message by processing that element. The message element is processed by removing it from the body and recording the error condition if no error condition has previously been recorded.

The function $remove\_entry$ removes a specified entry from a sequence.

$$[X]$$

$$remove\_entry : \mathbb{N} \times \operatorname{seq} X \longrightarrow \operatorname{seq} X$$

$$\forall\, i : \mathbb{N};\ x\_seq : \operatorname{seq} X \bullet$$
$$\quad remove\_entry(i, x\_seq) = ((1 \mathrel{..} i) \upharpoonright x\_seq) \frown (((i+1) \mathrel{..} \#x\_seq) \upharpoonright x\_seq)$$

Using this function, the processing of a data element with an invalid type is as follows:

$ProcessInvalidType$

$GeneralSendProcessing$

$$(message, i) \in Invalid\_msg\_types(\theta Mach)$$
$$int\_msg_2.body = remove\_entry(i, int\_msg_1.body)$$
$$int\_msg_2.status = int\_msg_1.status$$
$$int\_msg_1.error = \varnothing \Rightarrow$$
$$\quad int\_msg_2.error = \{\, Msg\_error\_invalid\_type \,\}$$

If a message contains port rights that have not yet been processed, then progress can be made in processing the message by processing one of the port rights. The first case to consider is that in which the name being processed does not denote a right appropriate for the type of transfer requested. In this case, $GeneralSendProcessing2$ resolves the name to $Null\_port$. Thus, the processing for this case can be specified as follows:

$ProcessRightBad$

$GeneralSendProcessing2$

$$(message, i, j) \in Unprocessed\_rights(\theta Mach)$$
$$port = Null\_port$$
$$int\_msg_2.status = int\_msg_1.status$$
$$int\_msg_1.error = \varnothing \Rightarrow$$
$$int\_msg_2.error = \{\, Msg\_error\_invalid\_right \,\}$$

In other words, the invalid right is replaced by a right for $Null\_port$. The conversion from an entry of type V_data to V_port makes progress towards completion of the request since there is one less unprocessed right in the resulting state.

The only differences between this and the processing of a valid port right are:

- it is not necessary to record an error for a valid port right
- the task's port name space and port's make-send count must be updated

$$
\begin{array}{|l}
\hline
\textit{ProcessRightGood} \\
\textit{GeneralSendProcessing2} \\
\hline
(message, i, j) \in Unprocessed\_rights(\theta Mach) \\
port \neq Null\_port \\
\underline{p}ort\_right\_rel' = Process\_right(\{ mach\_msg\_type \}, task, name, \underline{p}ort\_right\_rel) \\
\underline{m}ake\_send\_count' = \underline{m}ake\_send\_count \oplus \{ port \mapsto \\
\quad \underline{m}ake\_send\_count\ port + \#(\{ mach\_msg\_type \} \cap \{ Mmt\_make\_send \}) \} \\
int\_msg_2.status = int\_msg_1.status \wedge \\
int\_msg_2.error = int\_msg_1.error \\
\hline
\end{array}
$$

The case in which an out-of-line memory region is inaccessible to the sending task is specified as follows:

$$
\begin{array}{|l}
\hline
\textit{ProcessMemoryBad} \\
\textit{GeneralSendProcessing} \\
\hline
(message, i) \in Unprocessed\_memories(\theta Mach) \\
int\_msg_1.body(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)) \\
((task, page\_index) \notin \underline{a}llocated \vee \\
\quad Read \notin protection(task, page\_index)) \\
int\_msg_2.body = remove\_entry(i, int\_msg_1.body) \\
int\_msg_2.status = int\_msg_1.status \wedge \\
int\_msg_1.error = \emptyset \Rightarrow \\
\quad int\_msg_2.error = \{ Msg\_error\_invalid\_memory \} \\
\hline
\end{array}
$$

If an out-of-line memory region is accessible and does not carry any port rights, then the element of form Msg_region can be converted into an element of form $Transit\_memory$.

$$
\begin{array}{|l}
\hline
\textit{ProcessMemoryGood} \\
\textit{GeneralSendProcessing} \\
\hline
(message, i) \in Unprocessed\_memories(\theta Mach) \\
int\_msg_1.body(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)) \\
(task, page\_index) \in \underline{a}llocated \\
Read \notin protection(task, page\_index) \\
(mach\_msg\_type \notin Recognized\_transfer\_options \vee \\
\quad Co\_carries\_rights \notin int\_msg_1.header.complex) \\
((task, page\_index), (memory, offset)) \in \underline{m}ap\_rel \\
(\mathbf{let}\ inte == Transit\_memory(n, mach\_msg\_type, (task, memory, offset)) \bullet \\
\quad int\_msg_2.body = replace\_entry(i, inte, int\_msg_1.body)) \\
\underline{m}ap\_rel' = \mathbf{if}\ olsd = Msg\_deallocate \\
\quad \mathbf{then}\ \{ (task, page\_index) \} \lhd \underline{m}ap\_rel\ \mathbf{else}\ \underline{m}ap\_rel \\
int\_msg_2.status = int\_msg_1.status \wedge \\
int\_msg_1.error = int\_msg_2.error \\
\hline
\end{array}
$$

Note that the transferred memory is deallocated from the address space of the sending task if $olsd$ indicates that it should be deallocated.

If an out-of-line memory region is accessible, carries port rights, and is currently in memory, then the element of form Msg_region can be converted into an element of form Msg_value. The resulting element will subsequently be processed by $ProcessRightGood$ or $ProcessRightBad$.

The function Va_offset is used to add an integer to a virtual address.

$$Va\_offset : VIRTUAL\_ADDRESS \times \mathbb{N} \twoheadrightarrow VIRTUAL\_ADDRESS$$

The function $Index\_to\_offset$ is used to convert a page index to a page offset.

$$Index\_to\_offset : PAGE\_INDEX \rightarrowtail\!\!\!\!\twoheadrightarrow PAGE\_OFFSET$$

The function $Word\_to\_data$ is used to convert a word on a page to a data item.

$$Word\_to\_data : WORD \rightarrowtail MSG\_DATA$$

ProcessAvailableOutOfLineRights

$$
\begin{array}{l}
GeneralSendProcessing \\
\hline
(message, i) \in Unprocessed\_memories(\theta Mach) \\
int\_msg_1.body(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)) \\
mach\_msg\_type \in Recognized\_transfer\_options \\
Co\_carries\_rights \in int\_msg_1.header.complex \\
\exists\, page\_reference\_set : \mathbb{P}(\mathbb{N} \times VIRTUAL\_ADDRESS \times PAGE\_INDEX) \bullet \\
\quad page\_reference\_set = \{\, m : \mathbb{N};\ va_1 : VIRTUAL\_ADDRESS; \\
\quad page\_index_1 : PAGE\_INDEX \mid \\
\qquad m \in 1 \ldots n \land \\
\qquad va_1 = Va\_offset(va, m) \land \\
\qquad page\_index_1 = Address\_to\_index\ va_1 \,\} \land \\
\quad value\_seq_2 = \{k : \mathbb{N};\ va_2 : VIRTUAL\_ADDRESS; \\
\qquad page : PAGE;\ page\_offset : PAGE\_OFFSET;\ word : WORD; \\
\qquad page\_index_2 : PAGE\_INDEX;\ memory_2 : MEMORY;\ offset_2 : OFFSET; \\
\qquad msg\_data : MSG\_DATA \mid \\
\qquad\quad (k, va_2, page\_index_2) \in page\_reference\_set \land \\
\qquad\quad page\_index_2 = Address\_to\_index\ va_2 \land \\
\qquad\quad (task, page\_index_2) \in \underline{a}llocated \land \\
\qquad\quad Read \in protection(task, page\_index_2) \land \\
\qquad\quad ((task, page\_index_2), (memory_2, offset_2)) \in \underline{m}ap\_rel \land \\
\qquad\quad (page, (memory_2, offset_2)) \in \underline{r}epresents\_rel \land \\
\qquad\quad page\_offset = Index\_to\_offset\ page\_index_2 \land \\
\qquad\quad ((page, page\_offset), word) \in \underline{p}age\_word\_rel \land \\
\qquad\quad msg\_data = Word\_to\_data\ word \bullet \\
\qquad\qquad (k, V\_data(msg\_data, V\_data\_out)) \,\} \land \\
\quad (\mathbf{let}\ inte == Msg\_value(n, mach\_msg\_type, (task, value\_seq_2)) \bullet \\
\qquad int\_msg_2.body = replace\_entry(i, inte, int\_msg_1.body)) \land \\
\quad \underline{m}ap\_rel' = \mathbf{if}\ olsd = Msg\_deallocate \\
\qquad \mathbf{then}\ \{\, r : \mathbb{N};\ va_3 : VIRTUAL\_ADDRESS;\ page\_index_3 : PAGE\_INDEX \mid \\
\qquad\quad (r, va_3, page\_index_3) \in page\_reference\_set \bullet \\
\qquad\qquad (task, page\_index_3) \,\} \mathbin{\lhd\!\!\!-} \underline{m}ap\_rel \\
\qquad \mathbf{else}\ \underline{m}ap\_rel \\
int\_msg_2.status = int\_msg_1.status \land \\
int\_msg_1.error = int\_msg_2.error
\end{array}
$$

The set $page\_reference\_set$ denotes the pages that are referenced by the message element. If each page is in memory, the necessary data can be read from the pages and stored in $value\_seq_2$. Note that if $olsd$ indicates that the region should be deallocated, then each of the referenced pages is removed from the address space of the sending task.

If a page referenced by the message element is not accessible, then the processing is analogous to that described by $ProcessMemoryBad$.

$$
\begin{array}{l}
\textit{ProcessRightsMemoryBad} \\
\hline
\textit{GeneralSendProcessing} \\
\hline
(message, i) \in Unprocessed\_memories(\theta Mach) \\
int\_msg_1.body(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)) \\
mach\_msg\_type \in Recognized\_transfer\_options \\
Co\_carries\_rights \in int\_msg_1.header.complex \\
\exists\, page\_reference\_set : \mathbb{P}(\mathbb{N} \times VIRTUAL\_ADDRESS \times PAGE\_INDEX) \bullet \\
\quad page\_reference\_set = \{\, m : \mathbb{N};\ va_1 : VIRTUAL\_ADDRESS; \\
\quad page\_index_1 : PAGE\_INDEX \mid \\
\qquad m \in 1 \mathrel{..} n \wedge \\
\qquad va_1 = Va\_offset(va, m) \wedge \\
\qquad page\_index_1 = Address\_to\_index\ va_1 \,\} \wedge \\
\quad (\exists\, k : \mathbb{N};\ va_2 : VIRTUAL\_ADDRESS; \\
\quad page\_index_2 : PAGE\_INDEX \bullet \\
\qquad (k, va_2, page\_index_2) \in page\_reference\_set \wedge \\
\qquad page\_index_2 = Address\_to\_index\ va_2 \wedge \\
\qquad ((task, page\_index_2) \notin \underline{a}llocated \vee \\
\qquad\quad Read \notin protection(task, page\_index_2))) \wedge \\
\quad \underline{m}ap\_rel' = \mathbf{if}\ olsd = Msg\_deallocate \\
\qquad \mathbf{then}\ \{\, r : \mathbb{N};\ va_3 : VIRTUAL\_ADDRESS;\ page\_index_3 : PAGE\_INDEX \mid \\
\qquad\quad (r, va_3, page\_index_3) \in page\_reference\_set \bullet \\
\qquad\qquad (task, page\_index_3) \,\} \ntriangleleft \underline{m}ap\_rel \\
\qquad \mathbf{else}\ \underline{m}ap\_rel \\
int\_msg_2.body = remove\_entry(i, int\_msg_1.body) \\
int\_msg_2.status = int\_msg_1.status \\
int\_msg_1.error = \emptyset \Rightarrow \\
\quad int\_msg_2.error = \{\, Msg\_error\_invalid\_memory \,\} \\
\hline
\end{array}
$$

If a page referenced by the message element is accessible but is not in memory, then the kernel must request the page's data from the page's memory manager.

The following function is used to build the header for the request sent to the memory manager.

$$
\begin{array}{l}
Mach\_object\_data\_request : OPERATION \\
Build\_data\_request\_header : PORT \times PORT \longrightarrow MachInternalHeader \\
\hline
\forall\, port_1, port_2 : PORT;\ int\_msgh : MachInternalHeader \mid \\
\quad Build\_data\_request\_header(port_1, port_2) = int\_msgh \bullet \\
\qquad int\_msgh.local\_rights = \{\, Mmt\_copy\_send \,\} \wedge \\
\qquad int\_msgh.remote\_rights = Mmt\_make\_send\_once \wedge \\
\qquad int\_msgh.complex = \emptyset \wedge \\
\qquad int\_msgh.remote\_port = port_1 \wedge \\
\qquad int\_msgh.local\_port = \{port_2\} \wedge \\
\qquad int\_msgh.operation = Mach\_object\_data\_request
\end{array}
$$

The first port parameter is the remote port and the second one is the local port. The operation is specified as being $Mach\_object\_data\_request$.

The following functions are used to build the body for the request sent to the memory manager.

$$
\begin{array}{l}
Mmt\_integer : MACH\_MSG\_TYPE \\
Mmt\_protection : MACH\_MSG\_TYPE \\
Integer\_to\_data : \mathbb{N} \longrightarrow MSG\_DATA \\
Protection\_to\_data : \mathbb{P}\ PROTECTION \longrightarrow MSG\_DATA \\
Build\_data\_request\_body : TASK \times \mathbb{N} \times \mathbb{N} \times \mathbb{P}\ PROTECTION \longrightarrow INTERNAL\_BODY \\
\hline
\forall\, task : TASK;\ i, j : \mathbb{N};\ prot\_set : \mathbb{P}\ PROTECTION \bullet \\
\quad Build\_data\_request\_body(task, i, j, prot\_set) = \\
\quad\quad (\mathbf{let}\ value\_seq_1 == \\
\langle V\_data(Integer\_to\_data\ i,\ V\_data\_in),\ V\_data(Integer\_to\_data\ j,\ V\_data\_in)\rangle; \\
\quad\quad value\_seq_2 == \langle V\_data(Protection\_to\_data\ prot\_set,\ V\_data\_in)\rangle \bullet \\
\quad\quad\quad \langle Msg\_value(2, Mmt\_integer, (task, value\_seq_1)), \\
\quad\quad\quad Msg\_value(1, Mmt\_protection, (task, value\_seq_2))\rangle)
\end{array}
$$

The integers indicate, respectively, the desired offset in the memory object and length of the data. The set of protections specify the access modes desired for the object.

The following function is used to build the request sent to the memory manager.

$$
\begin{array}{l}
Build\_data\_request : TASK \times PORT \times PORT \times \mathbb{N} \times \mathbb{N} \times \mathbb{P}\ PROTECTION \longrightarrow \\
\quad InternalMessage \\
\hline
\forall\, task : TASK;\ port_1, port_2 : PORT;\ i, j : \mathbb{N};\ prot\_set : \mathbb{P}\ PROTECTION; \\
int\_msg : InternalMessage \mid \\
\quad int\_msg = Build\_data\_request(task, port_1, port_2, i, j, prot\_set) \bullet \\
\quad\quad int\_msg.header = Build\_data\_request\_header(port_1, port_2) \land \\
\quad\quad int\_msg.body = Build\_data\_request\_body(task, i, j, prot\_set) \land \\
\quad\quad int\_msg.option = \{\ Mach\_send\_msg\ \} \land \\
\quad\quad int\_msg.time\_out\_at = \varnothing \land \\
\quad\quad int\_msg.status = Msg\_stat\_send \land \\
\quad\quad int\_msg.error = \varnothing
\end{array}
$$

The function $Index\_to\_nat$ is used to convert a page index to an integer.

$$
Index\_to\_nat : PAGE\_INDEX \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}
$$

The constant $Page\_size$ denotes the number of words on a page.

$$
Page\_size : \mathbb{N}
$$

Using these definitions, the sending of a request to the memory manager can be specified as follows:

$$
\begin{array}{l}
\textit{RequestRightsData} \\
\hline
\textit{GeneralSendProcessing} \\
int\_msg! : InternalMessage \\
\hline
(message, i) \in Unprocessed\_memories(\theta Mach) \\
int\_msg_1.body(i) = Msg\_region(n, mach\_msg\_type, (task, va, olsd)) \\
mach\_msg\_type \in Recognized\_transfer\_options \\
Co\_carries\_rights \in int\_msg_1.header.complex \\
\exists\, page\_reference\_set : \mathbb{P}(\mathbb{N} \times VIRTUAL\_ADDRESS \times PAGE\_INDEX) \bullet \\
\quad page\_reference\_set = \{ m : \mathbb{N}; va_1 : VIRTUAL\_ADDRESS; \\
\quad page\_index_1 : PAGE\_INDEX \mid \\
\qquad m \in 1 \mathrel{..} n \land \\
\qquad va_1 = Va\_offset(va, m) \land \\
\qquad page\_index_1 = Address\_to\_index\ va_1 \} \land \\
\quad (\exists\, k : \mathbb{N}; page : PAGE; va_2 : VIRTUAL\_ADDRESS; \\
\quad page\_index_2 : PAGE\_INDEX; memory_2 : MEMORY; offset_2 : OFFSET \bullet \\
\qquad (k, va_2, page\_index_2) \in page\_reference\_set \land \\
\qquad page\_index_2 = Address\_to\_index\ va_2 \land \\
\qquad (task, page\_index_2) \in \underline{a}llocated \land \\
\qquad Read \in protection(task, page\_index_2) \land \\
\qquad ((task, page\_index_2), (memory_2, offset_2)) \in \underline{m}ap\_rel \land \\
\qquad (page, (memory_2, offset_2)) \notin \underline{r}epresents\_rel \land \\
\qquad int\_msg! = \\
\qquad\quad (\mathbf{let}\ port_1 == object\_port\ memory_2; \\
\qquad\quad port_2 == control\_port\ memory_2; \\
\qquad\quad prot\_set == \{ Read, Write, Execute \}; \\
\qquad\quad r == Index\_to\_nat\ page\_index_2; \\
\qquad\quad s == max\{ Page\_size, n - (k-1) * Page\_size \} \bullet \\
\qquad\qquad Build\_data\_request(\underline{k}ernel, port_1, port_2, r, s, prot\_set))) \\
int\_msg_2.body = remove\_entry(i, int\_msg_1.body) \\
int\_msg_2.status = int\_msg_1.status \\
int\_msg_1.error = int\_msg_2.error \\
\hline
\end{array}
$$

Note that $int\_msg!$ denotes the message that should be sent to the memory manager. The "sending" of this message would be represented by adding it to the range of $\underline{m}sg\_contents$. For simplicity, we do not address that processing here.

If the destination port for a message that has been processed does not exist, then the message can be discarded.

Editorial Note:
The model of the state component $\underline{f}orcibly\_queued$ was previously as a function from a port to a message. This model was based upon the Kernel Principles document which states "mach_msg provides an option allowing one message to be left waiting to be queued." However, this is not one message per port, but one message per port right. The model has now been fixed, but its ramifications on this section have not been determined. Therefore all mention of $\underline{f}orcibly\_queued$ within the Z has been commented out in this section (though none of the text has been changed).

CAR 4041 has been filed to address this issue.

$$
\begin{array}{l}
\textit{PortDied} \\
\hline
\Delta\ \textit{DtosExec} \\
\textit{message} : \textit{MESSAGE} \\
\hline
\textit{message} \in \textit{Processed\_messages}(\theta \textit{Mach}) \\
(\underline{m}\textit{sg\_contents message}).\textit{header}.\textit{remote\_port} \notin \underline{p}\textit{ort\_exists} \\
\underline{m}\textit{essage\_exists}' = \underline{m}\textit{essage\_exists} \setminus \{\ \textit{message}\ \}
\end{array}
$$

A message that has been processed can be queued at its destination port if that port exists and there is room in the message queue associated with the port or if the message was sent using a send-once right. The return status is as defined by the *error* component of the message if it is nonempty. Otherwise, the status is $Mm\_success$.

The function $Error\_to\_status$ converts an element of type MSG_ERROR to an element of type $MACH\_MSG\_RETURN$.

$$
\begin{array}{l}
\textit{Error\_to\_status} : \mathbb{P}\ \textit{MSG\_ERROR} \longrightarrow \textit{MACH\_MSG\_RETURN} \\
\hline
\textit{Error\_to\_status} = \{\ \{\ \textit{Msg\_error\_invalid\_memory}\ \} \mapsto \textit{Mm\_send\_invalid\_memory}, \\
\quad \{\ \textit{Msg\_error\_invalid\_right}\ \} \mapsto \textit{Mm\_send\_invalid\_right}, \\
\quad \{\ \textit{Msg\_error\_invalid\_type}\ \} \mapsto \textit{Mm\_send\_invalid\_type}, \\
\quad \{\ \textit{Msg\_error\_msg\_too\_small}\ \} \mapsto \textit{Mm\_send\_msg\_too\_small}\ \}
\end{array}
$$

$$
\begin{array}{l}
\textit{EnqueueMsg} \\
\hline
\Delta\ \textit{DtosExec} \\
\textit{message} : \textit{MESSAGE} \\
\textit{msg\_return}! : \textit{MACH\_MSG\_RETURN} \\
\hline
\textit{message} \in \textit{Processed\_messages}(\theta \textit{Mach}) \\
(\mathbf{let}\ \textit{port} == (\underline{m}\textit{sg\_contents message}).\textit{header}.\textit{remote\_port} \bullet \\
\quad \textit{port} \in \underline{p}\textit{ort\_exists} \land \\
\quad (\underline{q}\textit{\_limit port} > \textit{port\_size port} \lor \\
\quad\quad (\underline{m}\textit{sg\_contents message}).\textit{header}.\textit{remote\_rights} \in \\
\quad\quad\quad \{\ \textit{Mmt\_move\_send\_once}, \textit{Mmt\_make\_send\_once}\ \}) \land \\
\quad \underline{m}\textit{essage\_in\_port\_rel}' = \underline{m}\textit{essage\_in\_port\_rel} \oplus \{\ \textit{port} \mapsto \\
\quad\quad \underline{m}\textit{essage\_in\_port\_rel port} \frown \langle \textit{message} \rangle\ \}) \\
\textit{msg\_return}! = \\
\quad (\mathbf{let}\ \textit{msg\_error} == (\underline{m}\textit{sg\_contents message}).\textit{error} \bullet \\
\quad\quad \mathbf{if}\ \textit{msg\_error} \neq \varnothing\ \mathbf{then}\ \textit{Error\_to\_status msg\_error} \\
\quad\quad\quad \mathbf{else}\ \textit{Mm\_success})
\end{array}
$$

If the following conditions hold:

- The message was sent using a send right rather than a send-once right.
- The client specified the $Mach\_send\_notify$ option and the message either does not have a time out specified or the time out period has passed.
- The destination port exists and has a full message queue.
- No message is currently forcibly enqueued at the port.

then the message can be forcibly enqueued at the port.

If there already is a message forcibly queued at the port, then an error message is returned and a pseudo-receive is initiated. We represent that a pseudo receive has been initiated by changing the message status from Msg_stat_send to Msg_stat_pseudo.

If a time out was specified and the time out period has passed, then the message can time out with a pseudo receive operation being generated.

$$
\begin{array}{l}
\textbf{MsgSendTimeOut} \\
\hline
\Delta\ DtosExec \\
message : MESSAGE \\
msg\_return! : MACH\_MSG\_RETURN \\
\hline
message \in Processed\_messages(\theta Mach) \\
\exists\, i : \mathbb{N} \mid i \in (\underline{m}sg\_contents\ message).time\_out\_at \bullet \\
i \leq \underline{h}ost\_time \\
Mach\_send\_notify \in (\underline{m}sg\_contents\ message).option \\
(\underline{m}sg\_contents'\ message).option = (\underline{m}sg\_contents\ message).option \\
(\underline{m}sg\_contents'\ message).time\_out\_at = (\underline{m}sg\_contents\ message).time\_out\_at \\
(\underline{m}sg\_contents'\ message).status = Msg\_stat\_pseudo \\
(\underline{m}sg\_contents'\ message).error = \\
\qquad \textbf{if}\ (\underline{m}sg\_contents\ message).error = \varnothing\ \textbf{then}\ \{\ Msg\_error\_timed\_out\ \} \\
\qquad \textbf{else}\ (\underline{m}sg\_contents\ message).error
\end{array}
$$

C.2.2 Message Receive

The **mach_msg** request can be used to receive a message by including $Mach\_rcv\_msg$ in $option?$ and not including $Mach\_send\_msg$.

$$
\begin{array}{l}
\textbf{MachMsgRcv} \\
\hline
MachMsgSignature \\
\hline
Mach\_rcv\_msg \in option? \\
Mach\_send\_msg \notin option?
\end{array}
$$

C.2.2.1 Initial Processing

We use the following schema to describe receive operations that are processed as no-ops due to error conditions that arise during the initial processing of the request:

$$
\begin{array}{l}
\textbf{MachMsgRcvNoOp} \\
\hline
\Xi\ Dtos \\
MachMsgRcv
\end{array}
$$

If $rcv\_name?$ does not denote a receive right or a port set for the client task, then an error message is returned and no further processing occurs.

$$
\begin{array}{l}
\textbf{MachMsgRcvInvalidName} \\
\hline
MachMsgNoOp \\
\hline
(owning\_task\ client?, rcv\_name?) \notin (r\_right \cup port\_set\_namep) \\
msg\_return! = Mm\_rcv\_invalid\_name
\end{array}
$$

Otherwise, if $rcv\_name?$ is a member of a port set, then an error message is returned and no further processing occurs.

$$
\begin{array}{l}
\textbf{MachMsgRcvValidName} \\
\hline
Mach \\
client? : THREAD \\
rcv\_name? : NAME \\
\hline
(owning\_task\ client?, rcv\_name?) \in r\_right \cup port\_set\_namep
\end{array}
$$

$$
\begin{array}{l}
\textbf{MachMsgRcvInSet} \\
\hline
MachMsgRcvValidName \\
MachMsgRcvNoOp \\
\hline
(owning\_task\ client?, rcv\_name?) \in port\_set\_namep \\
msg\_return! = Mm\_rcv\_in\_set
\end{array}
$$

Otherwise, the request is queued at the end of the list of pending receives.

$$
\begin{array}{l}
\textbf{MachMsgRcvNotInSet} \\
\hline
MachMsgRcvValidName \\
\hline
(owning\_task\ client?, rcv\_name?) \notin port\_set\_namep
\end{array}
$$

$$
\begin{array}{l}
\textbf{MachMsgRcvMakePending} \\
\hline
MachMsgRcv \\
MachMsgRcvNotInSet \\
\hline
\forall p\_rcv : PendingReceive \mid \\
\quad p\_rcv.notify = notify? \land \\
\quad p\_rcv.option = option? \land \\
\quad p\_rcv.rcv\_size = rcv\_size? \land \\
\quad p\_rcv.time\_out\_at = \textbf{if } Mach\_rcv\_timeout \in option? \\
\qquad \textbf{then } \varnothing \textbf{ else } \{ time\_out? + \underline{h}ost\_time \} \bullet \\
\qquad \underline{p}ending\_receives' = \underline{p}ending\_receives \oplus \\
\qquad\quad \{ (owning\_task\ client?, rcv\_name?) \mapsto \\
\qquad\qquad \underline{p}ending\_receives(owning\_task\ client?, rcv\_name?) \frown \langle p\_rcv \rangle \}
\end{array}
$$

C.2.2.2 Kernel Processing

Only the first request in the sequence associated with a port can be processed when a message is detected at the port. We introduce the following schema to denote processing of the first request.

$$
\begin{array}{l}
\textbf{GeneralRcvProcessing} \\
\hline
\Delta\ DtosExec \\
p\_rcv : PendingReceive \\
task : TASK \\
name : NAME \\
\hline
(task, name) \in local\_namep \cap \operatorname{dom} \underline{p}ending\_receives \\
\#(\underline{p}ending\_receives\,(task, name)) \neq 0 \\
(\underline{p}ending\_receives\,(task, name))(1) = p\_rcv
\end{array}
$$

The components $task$ and $name$ denote the task that initiated the receive operation and the $name$ that $task$ specified as $rcv\_name?$. We require that $task$ is an existing task, $name$ is a name that is in use in $task$'s name space, there is a sequence of pending receive requests associated with $(task, name)$, and the sequence of requests is nonempty. We introduce the component p_rcv to denote the first request in the sequence.

For a receive operation to be successful, the name specified by the client must either be a receive right or the name of a port set. The following schema defines the processing for the case in which the name is neither a receive right nor a port set:

$$
\begin{array}{l}
\textbf{RcvPortDied} \\
\hline
GeneralRcvProcessing \\
msg\_return! : MACH\_MSG\_RETURN \\
\hline
(task, name) \notin (r\_right \cup port\_set\_namep) \\
\underline{p}ending\_receives' = \underline{p}ending\_receives \oplus \{ (task, name) \mapsto \\
\quad tail(\underline{p}ending\_receives(task, name)) \} \\
msg\_return! = Mm\_rcv\_port\_died
\end{array}
$$

The request also fails if the specified name is a receive right that belongs to a port set.

$$
\begin{array}{l}
\textbf{RcvPortChanged} \\
\hline
GeneralRcvProcessing \\
msg\_return! : MACH\_MSG\_RETURN \\
\hline
(task, name) \in r\_right \\
\exists\, name_1 : NAME \bullet \\
\quad (task, name_1) \in port\_set\_namep \land \\
\quad named\_port(task, name) \in port\_set(task, name_1) \\
\underline{p}ending\_receives' = \underline{p}ending\_receives \oplus \{ (task, name) \mapsto \\
\quad tail(\underline{p}ending\_receives(task, name)) \} \\
msg\_return! = Mm\_rcv\_port\_changed
\end{array}
$$

The following schemas define the negation of the previous checks.

$$
\begin{array}{l}
\textbf{GeneralRcvProcessing2} \\
\hline
GeneralRcvProcessing \\
name_1 : NAME \\
port : PORT \\
message : MESSAGE \\
int\_msg_1 : InternalMessage \\
\hline
(((\exists\, i : \mathbb{N} \bullet \\
\qquad (task, port, name, Receive, i) \in \underline{p}ort\_right\_rel) \land \\
\qquad name_1 = name) \lor \\
\quad ((task, name) \in port\_set\_namep \land \\
\qquad port \in port\_set(task, name) \land \\
\qquad named\_port(task, name_1) = port)) \\
(Mach\_rcv\_large \notin p\_rcv.option \lor \\
\quad (\underline{m}sg\_contents\ (\underline{m}essage\_in\_port\_rel\ port\ 1)).header.size \leq p\_rcv.rcv\_size) \\
\#(\underline{m}essage\_in\_port\_rel\ port) \neq 0 \\
(\underline{m}essage\_in\_port\_rel\ port)(1) = message \\
int\_msg_1 = \underline{m}sg\_contents\ message
\end{array}
$$

If $Mach\_rcv\_large$ is specified in $option?$ and the message to be received is larger than rcv_size, then an error message is returned, $msgh!.rcv\_size$ is set to the size of the message, and no further processing occurs.

$$\begin{array}{l}
\textit{MachMsgRcvTooLarge} \\
\hline
GeneralRcvProcessing2 \\
rcv_size! : \mathbb{N} \\
msg_return! : MACH_MSG_RETURN \\
\hline
Mach_rcv_large \in p_rcv.option \\
(\underline{m}sg_contents\ (\underline{m}essage_in_port_rel\ port\ 1)).header.size > p_rcv.rcv_size \\
\underline{p}ending_receives' = \underline{p}ending_receives \oplus \{ (task, name) \mapsto \\
\quad tail(\underline{p}ending_receives(task, name)) \} \\
rcv_size! = (\underline{m}sg_contents\ (\underline{m}essage_in_port_rel\ port\ 1)).header.size \\
msg_return! = Mm_rcv_too_large
\end{array}$$

$$\begin{array}{l}
\textit{GeneralRcvProcessing3} \\
\hline
GeneralRcvProcessing2 \\
int_msg_2 : InternalMessage \\
\hline
\underline{m}sg_contents' = \underline{m}sg_contents \oplus \{ message \mapsto int_msg_2 \} \\
int_msg_2.header = int_msg_1.header \\
int_msg_2.time_out_at = int_msg_1.time_out_at
\end{array}$$

Using this schema, we represent the initiation of the processing by setting the status of the message to Msg_stat_rcv.

$$\begin{array}{l}
\textit{InitiateMsgRcv} \\
\hline
GeneralRcvProcessing3 \\
\hline
int_msg_1.body = int_msg_2.body \\
int_msg_2.status = Msg_stat_rcv \\
int_msg_2.error = \varnothing
\end{array}$$

Subsequent processing occurs only on messages having a status ofMsg_stat_rcv. We use the following schema to represent processing of that form.

$$\begin{array}{l}
\textit{GeneralRcvProcessing4} \\
\hline
GeneralRcvProcessing2 \\
\hline
int_msg_1.status = Msg_stat_rcv
\end{array}$$

$$\begin{array}{l}
\textit{GeneralRcvProcessing5} \\
\hline
GeneralRcvProcessing3 \\
\hline
int_msg_1.status = Msg_stat_rcv
\end{array}$$

If the client did not specify $Mach\_rcv\_large$ and the message is larger than the specified receive size, then the message is dequeued and destroyed.

The following schema denotes the dequeueing and destruction of a message:

$$
\begin{array}{l}
\textit{DestroyMessage} \\
\hline
\Delta\ \textit{DtosExec} \\
\textit{message} : \textit{MESSAGE} \\
\textit{port} : \textit{PORT} \\
\textit{msg\_return}! : \textit{MACH\_MSG\_RETURN} \\
\hline
\textit{port} \in \operatorname{dom} \underline{m}\textit{essage\_in\_port\_rel} \\
\#(\underline{m}\textit{essage\_in\_port\_rel}\ \textit{port}) \neq 0 \\
(\underline{m}\textit{essage\_in\_port\_rel}\ \textit{port})(1) = \textit{message} \\
\underline{m}\textit{essage\_exists}' = \underline{m}\textit{essage\_exists} \setminus \{\ \textit{message}\ \} \\
\underline{m}\textit{sg\_contents}' = \{\ \textit{message}\ \} \lhd \underline{m}\textit{sg\_contents} \\
\underline{m}\textit{essage\_in\_port\_rel}' = \underline{m}\textit{essage\_in\_port\_rel} \oplus \{\ \textit{port} \mapsto \\
\qquad \textit{tail}(\underline{m}\textit{essage\_in\_port\_rel}\ \textit{port})\ \}
\end{array}
$$

Using this schema, the processing of a message that is too large can be specified as follows:

$$
\begin{array}{l}
\textit{MachMsgRcvTooLarge2} \\
\hline
\textit{GeneralRcvProcessing4} \\
\textit{DestroyMessage} \\
\hline
\textit{Mach\_rcv\_large} \notin \textit{p\_rcv.option} \\
\textit{int\_msg}_1.\textit{header.size} > \textit{p\_rcv.rcv\_size} \\
\textit{msg\_return}! = \textit{Mm\_rcv\_too\_large}
\end{array}
$$

If the client specified the $\textit{Mach\_rcv\_notify}$ option and the notify argument does not denote a valid receive right, then the processing is similar.

$$
\begin{array}{l}
\textit{MachMsgReceiveInvalidNotify} \\
\hline
\textit{GeneralRcvProcessing4} \\
\textit{DestroyMessage} \\
\hline
(\textit{Mach\_rcv\_large} \in \textit{p\_rcv.option} \lor \\
\qquad \textit{int\_msg}_1.\textit{header.size} \leq \textit{p\_rcv.rcv\_size}) \\
\textit{Mach\_rcv\_notify} \in \textit{p\_rcv.option} \\
(\textit{task}, \textit{p\_rcv.notify}) \notin \textit{r\_right} \\
\textit{msg\_return}! = \textit{Mm\_rcv\_invalid\_notify}
\end{array}
$$

C.2.3 Notes

In this section we describe aspects of the **mach_msg** processing that are not addressed in the preceding section and issues concerning the correctness of the specification. The main gaps in the current specification are the kernel processing of receive and pseudo-receive requests.

The majority of this processing is concerned with transforming a message from type $\textit{InternalMessage}$ to type $\textit{Message}$. To a large extent, this processing is simply the reverse of the processing described for the send request to transform a message from type $\textit{Message}$ to type $\textit{InternalMessage}$. There do not appear to be any major obstacles to defining this "reverse" processing.

A more serious problem with completing the specification of receive requests is the concurrency present in the system. For example, it is not clear from the available documentation what happens if a task loses the receive right for a port while the kernel is in the middle of dequeueing a message from that port. The kernel interface document states that a status of *Mm_rcv_port_died* is returned to the client, but is unclear about other aspects of the processing. In particular, it is not clear from the documentation whether transferred port rights become visible only after the kernel commits to dequeueing the message.

A related problem is addressing side-effects of send and receive operations. For example, when a receive operation dequeues and destroys a message, receive rights for ports can be destroyed. This requires the modeling of the destruction of the ports and the generation of notification messages that must be sent.

Another problem that is common to all of the specifications in the FTLS is that input and output parameters are represented by value while they are actually implemented as references. In reality, the client specifies a virtual address rather than specifying a message header and body. The kernel assumes that the message header starts at the specified address and that the message body starts directly after the message header. One example of the ramifications of this simplification in the specification is that the specification does not address the case in which the sender of a message does not have access to the memory containing the header or body. In this case, the implementation treats the request as a no-op and returns a status of *mm_send_invalid_data*. A more complicated example is that the specification does not address the case in which the memory indicated by the virtual address is not resident. In this case, the kernel must enter a dialogue with a memory manager to determine the message header and body to use for the request.

Several subtle aspects of Mach are unclear from the available documentation. Examples include:

- It is not clear what the types of the *remote_rights* and *local_rights* fields of the message header are. The specifications models them both as sets of *MACH_MSG_TYPE*. This means that a send or send-once right must be transferred to the receiving task. Without examining the Mach source code, it is not possible to tell whether this is really how Mach works. A related question is whether more than one type of right can be passed at a time. If not, then the sets of *MACH_MSG_TYPE* should be constrained to having at most one element.

- When determining whether an area of out-of-line memory is accessible by a client, it is not clear whether the client's access to all of the pages comprising the region must be checked or whether it suffices to simply check access to the first page. The latter would be more efficient, but it requires that the kernel ensure that a client have the same access to all pages comprising a memory object. Although this property is desirable, it is not yet captured in the FTLS state.

- For simplicity, the current specification assumes that all of the pages containing port rights passed in out-of-line memory must be resident before the kernel can process the rights. It is possible that the implementation allows the kernel to process the rights contained on resident pages while waiting for the data containing the other rights. Modeling this capability would slightly complicate the model since provision would have to be made for data elements that are partially in-line and partially out-of-line.

Appendix D

Refinements of the State Model

In this appendix we refine portions of the state model to a lower level of detail. This models some of the data types and relationships that are used to implement the high-level abstract model described in the Basic Kernel State Definition and DTOS State Extensions chapters.

D.1 Additional Z Extensions

We define a function Gen_set to model generic queues. This function will be used in refining many of the components of the state model. A generic queue has a head element that points to the first element of a linked list of queue elements. $HEAD$ is the generic type of the head element of the queue, and $ELEM$ is the generic type of the elements of the queue. If $head\_fnc$ maps the head of a queue to the first element of the queue and $next\_fnc$ maps a queue element to its successor, then the expression $Gen\_set(head\_fnc, next\_fnc)$ denotes a function mapping each element of $HEAD$ to the set of elements in its queue. We define a function Gen_seq to model generic sequences. This function will also be used in refining components of the state model. A generic sequence has a head element that points to the first element of a sequence of elements. The expression $Gen\_seq(head\_fnc, next\_fnc)$ denotes a function mapping each element of $HEAD$ to its sequence of elements.[18] Note that for certain values of $next\_fnc$ $Gen\_seq(head\_fnc, next\_fnc)$ may be infinite and therefore not of the type $\mathrm{seq}\ ELEM$.

$$
\begin{array}{l}
[HEAD, ELEM] \\
\hline
Gen\_set : (HEAD \pfun ELEM) \times (ELEM \pfun ELEM) \\
\quad \longrightarrow (HEAD \longrightarrow \mathbb{P}\ ELEM) \\
Gen\_seq : (HEAD \pfun ELEM) \times (ELEM \pfun ELEM) \\
\quad \longrightarrow (HEAD \longrightarrow (\mathbb{N}_1 \pfun ELEM)) \\
\hline
\forall\ head\_fnc : HEAD \pfun ELEM; \\
\quad next\_fnc : ELEM \pfun ELEM; \\
\quad head : HEAD \\
\bullet\ Gen\_seq(head\_fnc, next\_fnc)(head) \\
\quad = \{\ i : \mathbb{N}_1;\ e : ELEM \mid (head, e) \in head\_fnc \mathbin{;} (next\_fnc^{i-1})\ \} \\
\land\ Gen\_set(head\_fnc, next\_fnc)(head) \\
\quad = \mathrm{ran}(Gen\_seq(head\_fnc, next\_fnc)(head))
\end{array}
$$

D.2 Refinement of IPC Name Spaces

In refining the specification of IPC name spaces we introduce the following additional types:

$$
\begin{array}{l}
[\,IPC\_SPACE, IPC\_ENTRY, IPC\_OBJECT, PORT\_SET, IPC\_SPLAY\_TREE, \\
\quad IPC\_TREE\_NODE\,] \\
IPC\_TABLE == \mathbb{N}_1 \pfun IPC\_ENTRY
\end{array}
$$

[18] The Z expression $Q \mathbin{;} R$ denotes the composition of two relations with Q applied first followed by R. The expression R^k denotes the relation resulting from k applications of relation R. If $k = 0$, R^k denotes the identity relation. Thus, $Q \mathbin{;} (R^k)$ denotes one application of Q followed by k applications of R.

IPC_SPACE is the representation of a name space. Each name space consists of a set of elements of type IPC_ENTRY. Some of these entries point to an IPC_OBJECT which may be either a port or a port set. Note that we are introducing an explicit given type for port sets rather than representing them merely as a set of ports. This agrees with the prototype and makes it easier to model properties of port sets such as the message queue of a port set. The entries in a space are organized into two data structures, an IPC_TABLE and an IPC_SPLAY_TREE. An IPC_TABLE is simply a sequence of IPC_ENTRY that may have gaps in it. A splay tree is a search tree containing nodes of type IPC_TREE_NODE. Each IPC_TREE_NODE points to an IPC_ENTRY.

The expression $\underline{t}ask\_space(tk)$ denotes the IPC_SPACE associated with task tk. No two tasks have the same value for this expression. The set $\underline{s}pacep$ denotes the existing IPC name spaces. The expression $\underline{s}pace\_table(sp)$ denotes the IPC_TABLE associated with space sp, and $\underline{s}pace\_table\_size(sp)$ denotes the current maximum size of this table. Note that this value may change dynamically to improve performance and memory utilization. The expression $\underline{s}pace\_tree(sp)$ denotes the IPC_SPLAY_TREE associated with space sp. Every space has both a table and a splay tree although one or both of these could be empty.

$IpcSpace$

$TaskExist$
$\underline{t}ask\_space : TASK \rightarrowtail\!\!\!\!\!\!+\; IPC\_SPACE$
$\underline{s}pacep : \mathbb{P}\ IPC\_SPACE$
$\underline{s}pace\_table : IPC\_SPACE \twoheadrightarrow\!\!\!\!\!\!+\; IPC\_TABLE$
$\underline{s}pace\_table\_size : IPC\_SPACE \twoheadrightarrow\!\!\!\!\!\!+\; \mathbb{N}$
$\underline{s}pace\_tree : IPC\_SPACE \twoheadrightarrow\!\!\!\!\!\!+\; IPC\_SPLAY\_TREE$

$\mathrm{dom}\ \underline{t}ask\_space = \underline{t}ask\_exists$
$\mathrm{ran}\ \underline{t}ask\_space = \underline{s}pacep$
$\mathrm{dom}\ \underline{s}pace\_table = \mathrm{dom}\ \underline{s}pace\_table\_size = \mathrm{dom}\ \underline{s}pace\_tree = \underline{s}pacep$
$\forall sp : \underline{s}pacep \bullet max(\mathrm{dom}(\underline{s}pace\_table sp)) < \underline{s}pace\_table\_size(sp)$

We augment the set of rights to contain $Port\_set\_right$ and $Dead\_name\_right$. The former is the right associated with an entry for a port set, and the latter is a right that may be associated with a dead name. Because the marking of dead rights in Mach is performed lazily, a dead right need not be marked $Dead\_name\_right$. It is also recognized as dead if it points to an inactive IPC_OBJECT.

$$ALL\_RIGHTS ::= Right\_for\_port\langle\langle RIGHT\rangle\rangle \mid Port\_set\_right \mid Dead\_name\_right$$

$Receive\_right, Send\_right, Send\_once\_right : ALL\_RIGHTS$

$Receive\_right = Right\_for\_port(Receive)$
$Send\_right = Right\_for\_port(Send)$
$Send\_once\_right = Right\_for\_port(Send\_once)$

The set $\underline{e}ntryp$ denotes the existing IPC_ENTRY elements. An entry is marked with a generation that is used in determining whether it is out of date. The expression $\underline{e}ntry\_gen(entry)$ denotes the generation of $entry$. The expression $\underline{e}ntry\_object(entry)$ denotes the IPC_OBJECT associated with $entry$. The expression $\underline{e}ntry\_rights(entry)$ denotes the set of ALL_RIGHTS associated with $entry$. Finally, the expression $\underline{e}ntry\_count(entry)$ denotes the number of send rights denoted by $entry$ when a name denotes multiple send rights for a task. If $\underline{e}ntry\_count(entry)$ is positive, then $entry$ must denote a $Send\_right$, $Send\_once\_right$ or $Dead\_name\_right$.

Editorial Note:
Should probably add dead name notification requests to this.

$IpcEntry$

$$\underline{e}ntryp : \mathbb{P}\, IPC\_ENTRY$$
$$\underline{e}ntry\_gen : IPC\_ENTRY \rightarrowtail\!\!\!\!\!\!\!+ \;\mathbb{N}$$
$$\underline{e}ntry\_object : IPC\_ENTRY \nrightarrow IPC\_OBJECT$$
$$\underline{e}ntry\_rights : IPC\_ENTRY \nrightarrow \mathbb{P}\, ALL\_RIGHTS$$
$$\underline{e}ntry\_count : IPC\_ENTRY \nrightarrow \mathbb{N}$$

$$\mathrm{dom}\, \underline{e}ntry\_gen = \mathrm{dom}\, \underline{e}ntry\_object = \mathrm{dom}\, \underline{e}ntry\_rights = \mathrm{dom}\, \underline{e}ntry\_count = \underline{e}ntryp$$
$$\forall\, entry : IPC\_ENTRY \mid \underline{e}ntry\_count(entry) > 0 \bullet \underline{e}ntry\_rights(entry) \cap \{Send\_right, Send\_once\_right, Dead\_name\_right\} \neq \emptyset$$

Every entry in the table associated with a space must denote some right.

$IpcTableEntry$

$IpcSpace$
$IpcEntry$

$$\forall\, entry : IPC\_ENTRY;\ table : IPC\_TABLE \mid table \in \mathrm{ran}\, \underline{s}pace\_table \land entry \in \mathrm{ran}\, table \bullet \underline{e}ntry\_rights(entry) \neq \emptyset$$

The set $\underline{o}bjectp$ denotes the existing IPC_OBJECT elements. An existing object may be inactive. The active objects are denoted by $\underline{a}ctive\_objects$. An IPC_OBJECT may be either a port or a port set. The expressions $\underline{o}bject\_as\_port(obj)$ and $\underline{o}bject\_as\_port\_set(obj)$ denote the associated port or port set. The domains of these two functions partition the set of existing objects.

$IpcObject$

$$\underline{o}bjectp : \mathbb{P}\, IPC\_OBJECT$$
$$\underline{a}ctive\_objects : \mathbb{P}\, IPC\_OBJECT$$
$$\underline{o}bject\_as\_port : IPC\_OBJECT \rightarrowtail\!\!\!\!\!\!\!+\; PORT$$
$$\underline{o}bject\_as\_port\_set : IPC\_OBJECT \rightarrowtail\!\!\!\!\!\!\!+\; PORT\_SET$$

$$\underline{a}ctive\_objects \subseteq \underline{o}bjectp$$
$$\langle \mathrm{dom}\, \underline{o}bject\_as\_port, \mathrm{dom}\, \underline{o}bject\_as\_port\_set \rangle \;\mathsf{partition}\; \underline{o}bjectp$$

A port $port$ is in a port set P if and only if $(port, P) \in \underline{p}ort\_in\_set$.

$PortInSet$

$PortExist$
$$\underline{p}ort\_setp : \mathbb{P}\, PORT\_SET$$
$$\underline{p}ort\_in\_set : PORT \nrightarrow PORT\_SET$$

$$\mathrm{dom}\, \underline{p}ort\_in\_set \subseteq \underline{p}ort\_exists$$
$$\mathrm{ran}\, \underline{p}ort\_in\_set \subseteq \underline{p}ort\_setp$$

Each $NAME$ n encodes an index denoted $\underline{n}ame\_index(n)$ and a generation denoted $\underline{n}ame\_gen(n)$. If names n_1 and n_2 have the same index and generation, then they are the same name.

$$
\begin{array}{l}
\textit{Name} \\
\hline
\underline{n}ame\_index : NAME \longrightarrow \mathbb{N} \\
\underline{n}ame\_gen : NAME \longrightarrow \mathbb{N} \\
\hline
\forall n_1, n_2 : NAME \\
\mid \underline{n}ame\_index(n_1) = \underline{n}ame\_index(n_2) \\
\quad \wedge \underline{n}ame\_gen(n_1) = \underline{n}ame\_gen(n_2) \\
\bullet\ n_1 = n_2
\end{array}
$$

The set $\underline{s}play\_treep$ denotes the existing splay trees. For efficiency of lookup, a splay tree is represented internally by three (possibly empty) tree structures, a left, a right and a middle tree. The root IPC_TREE_NODE of each of these trees (when the tree is nonempty) is denoted, respectively, by $\underline{t}ree\_left(splay)$, $\underline{t}ree\_right(splay)$ and $\underline{t}ree\_middle(splay)$. A pair $(splay, node)$ is in $tree\_trees$ if and only if $node$ is the root of one of the three trees associated with $splay$.

$$
\begin{array}{l}
\textit{IpcSplayTree} \\
\hline
\underline{s}play\_treep : \mathbb{P}\ IPC\_SPLAY\_TREE \\
\underline{t}ree\_middle : IPC\_SPLAY\_TREE \rightarrow\!\!\!\!\!\!+\ IPC\_TREE\_NODE \\
\underline{t}ree\_left : IPC\_SPLAY\_TREE \rightarrow\!\!\!\!\!\!+\ IPC\_TREE\_NODE \\
\underline{t}ree\_right : IPC\_SPLAY\_TREE \rightarrow\!\!\!\!\!\!+\ IPC\_TREE\_NODE \\
tree\_trees : IPC\_SPLAY\_TREE \longleftrightarrow IPC\_TREE\_NODE \\
\hline
\mathrm{dom}\ tree\_trees \subseteq \underline{s}play\_treep \\
tree\_trees = \underline{t}ree\_middle \cup \underline{t}ree\_left \cup \underline{t}ree\_right
\end{array}
$$

The set $\underline{t}ree\_nodep$ denotes the set of existing IPC_TREE_NODE elements. Each tree node $node$ points to an IPC_ENTRY which is denoted $\underline{t}ree\_node\_entry(node)$. The expression $\underline{t}ree\_node\_name(node)$ denotes a $NAME$ associated with $node$. Each tree node may have a left and a right child tree node. These are denoted by the expressions $\underline{t}ree\_node\_lchild(node)$ and $\underline{t}ree\_node\_rchild(node)$. A pair $(node_1, node_2)$ is in $tree\_node\_children$ if and only if $node_2$ is either the left of right child of $node_1$.

$$
\begin{array}{l}
\textit{IpcTreeNode} \\
\hline
\underline{t}ree\_nodep : \mathbb{P}\ IPC\_TREE\_NODE \\
\underline{t}ree\_node\_entry : IPC\_TREE\_NODE \rightarrow\!\!\!\!\!\!+\ IPC\_ENTRY \\
\underline{t}ree\_node\_name : IPC\_TREE\_NODE \rightarrow\!\!\!\!\!\!+\ NAME \\
\underline{t}ree\_node\_lchild : IPC\_TREE\_NODE \rightarrow\!\!\!\!\!\!+\ IPC\_TREE\_NODE \\
\underline{t}ree\_node\_rchild : IPC\_TREE\_NODE \rightarrow\!\!\!\!\!\!+\ IPC\_TREE\_NODE \\
tree\_node\_children : IPC\_TREE\_NODE \longleftrightarrow IPC\_TREE\_NODE \\
\hline
\mathrm{dom}\ \underline{t}ree\_node\_entry = \mathrm{dom}\ \underline{t}ree\_node\_name = \underline{t}ree\_nodep \\
\mathrm{dom}\ \underline{t}ree\_node\_lchild \subseteq \underline{t}ree\_nodep \\
\mathrm{dom}\ \underline{t}ree\_node\_rchild \subseteq \underline{t}ree\_nodep \\
tree\_node\_children = \underline{t}ree\_node\_lchild \cup \underline{t}ree\_node\_rchild
\end{array}
$$

We are now ready to define the relations $\underline{p}ort\_right\_rel$, $\underline{p}ort\_set\_rel$, and $\underline{d}ead\_right\_rel$. All three share the requirement that the space of the task must contain an entry with the appropriate name. If the entry is in the table, then the index of the name must match the position of the name in the table, and the generations of the name and entry must be identical. If the entry is in the splay tree, then the name in the tree node must equal the given name. This requirement is abstracted by the relation $entry\_in\_space$. A triple $(task, name, entry)$ is in this relation if an only if it satisfies the above requirement.

$$
\begin{array}{l}
\textbf{EntryInSpace} \\
\hline
TaskExist \\
IpcTableEntry \\
Name \\
IpcSplayTree \\
IpcTreeNode \\
entry\_in\_space : \mathbb{P}(TASK \times NAME \times IPC\_ENTRY) \\
\hline
\forall tk : TASK; n : NAME; entry : IPC\_ENTRY \\
\bullet (tk, n, entry) \in entry\_in\_space \\
\Leftrightarrow (tk \in \underline{t}ask\_exists \\
\quad \land entry \in \underline{e}ntryp \\
\quad \land (((\underline{n}ame\_index(n), entry) \in \underline{s}pace\_table(\underline{t}ask\_space(tk)) \\
\qquad\qquad \land \underline{n}ame\_gen(n) = \underline{e}ntry\_gen(entry)) \\
\qquad \lor (\exists splay : IPC\_SPLAY\_TREE; \\
\qquad\qquad\quad tree\_nodes : \mathrm{seq}_1\, IPC\_TREE\_NODE \\
\qquad\qquad \bullet (\underline{t}ask\_space(tk), splay) \in \underline{s}pace\_tree \\
\qquad\qquad \land (splay, tree\_nodes(1)) \in tree\_trees \\
\qquad\qquad \land (last\ tree\_nodes, entry) \in \underline{t}ree\_node\_entry \\
\qquad\qquad \land (last\ tree\_nodes, n) \in \underline{t}ree\_node\_name \\
\qquad\qquad \land (\forall i : 2 \mathinner{..} \#tree\_nodes \\
\qquad\qquad\quad \bullet (tree\_nodes(i-1), tree\_nodes(i)) \in tree\_node\_children))))
\end{array}
$$

A 5-tuple $(tk, p, n, r, count)$ is in $\underline{p}ort\_right\_rel$ if and only if there exists an IPC_ENTRY, $entry$ such that

- $(tk, n, entry)$ is in $entry\_in\_space$,
- r is one of the rights associated with the entry all of which are rights to use a port (i.e., not port set rights nor dead rights),
- an active object is associated with the entry,
- the object is a port, and
- either
  - r is a receive right and $count$ is 1, or
  - r is not a receive right and $count$ is the right count of the entry.

$$\begin{array}{l}
\textit{PortRightRefinement} \\
\hline
\textit{TasksAndPorts} \\
\textit{EntryInSpace} \\
\textit{IpcObject} \\
\hline
\forall\, tk : TASK;\ p : PORT;\ n : NAME;\ r : RIGHT;\ count : \mathbb{N}_1 \\
\bullet\ (tk, p, n, r, count) \in \underline{p}ort_right_rel \\
\Leftrightarrow (\exists\, entry : IPC_ENTRY \\
\quad \bullet\ (tk, n, entry) \in entry_in_space \\
\quad \wedge\ Right_for_port(r) \in \underline{e}ntry_rights(entry) \subseteq \operatorname{ran} Right_for_port \\
\quad \wedge\ \underline{e}ntry_object(entry) \in \underline{a}ctive_objects \\
\quad \wedge\ (\underline{e}ntry_object(entry), p) \in \underline{o}bject_as_port \\
\quad \wedge\ ((r = Receive \wedge count = 1) \\
\qquad \vee\ (r \neq Receive \wedge count = \underline{e}ntry_count(entry))))
\end{array}$$

Editorial Note:
This is to cover up the name vs. port discrepancy in version 1.13 of the FTLS. When we incorporate this refinement is a new version of the FTLS in which this discrepancy is fixed, we must remove this schema and replace $\underline{n}ew\_port\_set$ below with $port\_set$.

$$\begin{array}{l}
\textit{NewPortSets} \\
\hline
\textit{TasksAndRights} \\
\underline{n}ew_port_set : (TASK \times NAME) \nrightarrow \mathbb{P}\, PORT
\end{array}$$

We define $\underline{p}ort\_set\_rel$ indirectly by defining $\underline{n}ew\_port\_set$. A port $port$ is in $\underline{n}ew\_port\_set(tk, n)$ if and only if there exists an IPC_ENTRY, $entry$ and a port set PS such that

- $(tk, n, entry)$ is in $entry\_in\_space$,
- the right associated with the entry is $Port\_set\_right$,
- an active object is associated with the entry,
- the object is PS and
- $port$ is an element of PS.

$$\begin{array}{l}
\textit{PortSetRefinement} \\
\hline
\textit{NewPortSets} \\
\textit{EntryInSpace} \\
\textit{IpcObject} \\
\textit{PortInSet} \\
\hline
\forall\, tk : TASK;\ port : PORT;\ n : NAME \\
\bullet\ port \in \underline{n}ew_port_set(tk, n) \\
\Leftrightarrow (\exists\, entry : IPC_ENTRY;\ PS : PORT_SET \\
\quad \bullet\ (tk, n, entry) \in entry_in_space \\
\quad \wedge\ \underline{e}ntry_rights(entry) = \{Port_set_right\} \\
\quad \wedge\ \underline{e}ntry_object(entry) \in \underline{a}ctive_objects \\
\quad \wedge\ (\underline{e}ntry_object(entry), PS) \in \underline{o}bject_as_port_set \\
\quad \wedge\ (port, PS) \in \underline{p}ort_in_set)
\end{array}$$

A triple $(tk, n, count)$ is in $\underline{d}ead\_right\_rel$ if and only if there exists an IPC_ENTRY, $entry$ such that

- $(tk, n, entry)$ is in $entry\_in\_space$,
- either the rights associated with the entry include $Dead\_name\_right$, or the entry points to an inactive object,
- $count$ is the right count of the entry,

$$
\begin{array}{l}
\textit{DeadRightRefinement} \\
\hline
DeadRights \\
EntryInSpace \\
IpcObject \\
\hline
\forall tk : TASK; n : NAME; count : \mathbb{N}_1 \\
\bullet (tk, n, count) \in \underline{d}ead\_right\_rel \\
\Leftrightarrow (\exists entry : IPC\_ENTRY \\
\quad \bullet (tk, n, entry) \in entry\_in\_space \\
\quad \land (Dead\_name\_right \in \underline{e}ntry\_rights(entry) \\
\qquad \lor \underline{e}ntry\_object(entry) \notin \underline{a}ctive\_objects) \\
\quad \land count = \underline{e}ntry\_count(entry))
\end{array}
$$

The schema $IpcRefinement$ defines the refinements for IPC name spaces.

$$
\begin{array}{l}
\textit{IpcRefinement} \\
\hline
PortRightRefinement \\
PortSetRefinement \\
DeadRightRefinement
\end{array}
$$

D.3 Refinement of Pending Receives

The expression $\underline{p}ort\_waiting\_threads\_head(port)$ denotes the first $THREAD$, if one exists, in the sequence of threads waiting to receive a message from $port$. The expression $\underline{p}ort\_set\_waiting\_threads\_head(pset)$ denotes the first $THREAD$, if one exists, in the sequence of threads waiting to receive a message from port set $pset$. The expression $\underline{n}ext\_waiting\_thread(th)$ denotes the successor $THREAD$ of th, if one exists, in the sequence of threads waiting to receive a message from the port or port set from which th is waiting to receive a message.

$$
\begin{array}{l}
\textit{WaitingThreads} \\
\hline
ThreadExist \\
PortExist \\
PortInSet \\
\underline{p}ort\_waiting\_threads\_head : PORT \nrightarrow THREAD \\
\underline{p}ort\_set\_waiting\_threads\_head : PORT\_SET \nrightarrow THREAD \\
\underline{n}ext\_waiting\_thread : THREAD \nrightarrow THREAD \\
\hline
\operatorname{dom} \underline{p}ort\_waiting\_threads\_head \subseteq \underline{p}ort\_exists \\
\operatorname{dom} \underline{p}ort\_set\_waiting\_threads\_head \subseteq \underline{p}ort\_setp \\
\operatorname{dom} \underline{n}ext\_waiting\_thread \subseteq \underline{t}hread\_exists
\end{array}
$$

The expression $\underline{t}hread\_pending\_receive(th)$ denotes the $PendingReceive$ data stored in a thread th that is waiting to receive a message.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StoredReceiveState}} \\
\hline
ThreadExist \\
\underline{t}hread\_pending\_receive : THREAD \rightarrow\!\!\!\!\!\!+ PendingReceive \\
\hline
\operatorname{dom} \underline{t}hread\_pending\_receive \subseteq \underline{t}hread\_exists \\
\hline
\end{array}
$$

The expression $named\_port\_set(tk, nm)$ denotes the port set named by nm in the IPC name space of task tk.

Editorial Note:
It might make more sense to have this in the regular state, not the refinements. Since it is here, we will refine it right away.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TasksAndPortSets}} \\
\hline
EntryInSpace \\
IpcObject \\
named\_port\_set : (TASK \times NAME) \rightarrow\!\!\!\!\!\!+ PORT\_SET \\
\hline
\forall\, tk : TASK;\ n : NAME;\ ps : PORT\_SET \\
\bullet\ ps = named\_port\_set(tk, n) \\
\Leftrightarrow (\exists\, entry : IPC\_ENTRY \\
\quad \bullet\ (tk, n, entry) \in entry\_in\_space \\
\quad \land\ \underline{e}ntry\_rights(entry) = \{Port\_set\_right\} \\
\quad \land\ \underline{e}ntry\_object(entry) \in \underline{a}ctive\_objects \\
\quad \land\ (\underline{e}ntry\_object(entry), ps) \in \underline{o}bject\_as\_port\_set) \\
\hline
\end{array}
$$

The expression $\underline{w}aiting\_for\_port(tk, nm)$ denotes the sequence of threads that are waiting for a message on the port named by nm in the IPC name space of tk. Note that $Gen\_seq(\underline{p}ort\_waiting\_threads\_head \,\text{⨾}\, \underline{n}ext\_waiting\_thread)$ denotes a function of type $PORT \rightarrow\!\!\!\!\!\!+ \operatorname{seq} THREAD$ where a thread th is in the sequence associated with a port p if and only if th is waiting to receive a message from p.[19] The expression $\underline{w}aiting\_for\_port\_set(tk, nm)$ denotes the sequence of threads that are waiting for a message on the port set named by nm in the IPC name space of tk. A name may not name both a port and a port set for the same task. Thus, the domains of $\underline{w}aiting\_for\_port$ and $\underline{w}aiting\_for\_port\_set$ are disjoint. For convenience, we define $waiting\_for\_message$ to be the union of the functions $\underline{w}aiting\_for\_port$ and $\underline{w}aiting\_for\_port\_set$. Because the two domains are disjoint, $waiting\_for\_message$ is necessarily a function. Every thread that is waiting for a message holds $PendingReceive$ information.

[19] The Z expression $Q \,\text{⨾}\, R$ denotes the composition of two relations with Q applied first followed by R.

$$
\begin{array}{l}
\textit{TasksAndWaiting} \\
\hline
\textit{WaitingThreads} \\
\textit{TasksAndPorts} \\
\textit{TasksAndPortSets} \\
\textit{StoredReceiveState} \\
\underline{w}aiting\_for\_port : TASK \times NAME \rightarrow\!\!\!\!\!\!+\; \mathrm{seq}\ THREAD \\
\underline{w}aiting\_for\_port\_set : TASK \times NAME \rightarrow\!\!\!\!\!\!+\; \mathrm{seq}\ THREAD \\
waiting\_for\_message : TASK \times NAME \rightarrow\!\!\!\!\!\!+\; \mathrm{seq}\ THREAD \\
\hline
\underline{w}aiting\_for\_port = named\_port \\
\quad \fatsemi Gen\_seq(\underline{p}ort\_waiting\_threads\_head, \underline{n}ext\_waiting\_thread) \\
\underline{w}aiting\_for\_port\_set = named\_port\_set \\
\quad \fatsemi Gen\_seq(\underline{p}ort\_set\_waiting\_threads\_head, \underline{n}ext\_waiting\_thread) \\
waiting\_for\_message = \underline{w}aiting\_for\_port \cup \underline{w}aiting\_for\_port\_set \\
\forall\, tk : TASK; n : NAME \\
\bullet\ \mathrm{ran}(waiting\_for\_message(tk, n)) \subseteq \mathrm{dom}\, \underline{t}hread\_pending\_receive
\end{array}
$$

We now refine the definition of $\underline{p}ending\_receives$. For any (tk, n) pair the sequence of $PendingReceive$ values associated with name n for task tk is found by extracting (via $\underline{t}hread\_pending\_receive$) the $PendingReceive$ data from each thread in the sequence $waiting\_for\_message(tk, n)$.

$$
\begin{array}{l}
\textit{PendingReceiveRefinement} \\
\hline
\textit{TasksAndWaiting} \\
\textit{Messages} \\
\hline
\forall\, tk : TASK; n : NAME \\
\bullet\ \underline{p}ending\_receives(tk, n) \\
\qquad = waiting\_for\_message(tk, n) \fatsemi \underline{t}hread\_pending\_receive
\end{array}
$$

D.4 Refinement of Virtual Memory

In refining the specification of virtual memory we introduce the following additional types:

$$[\, VM\_MAP, VM\_ENTRY, VM\_MAP\_OBJECT \,]$$

VM_MAP is the representation of a virtual address space. Each map consists of a sequence of elements of type VM_ENTRY. A VM_ENTRY denotes a contiguous range of virtual addresses that share the same properties (e.g., protections and inheritance options). Some of these entries point to a VM_MAP_OBJECT which may be either a memory object or a another memory map called a submap.

The expression $\underline{t}ask\_map(tk)$ denotes the VM_MAP associated with task tk. Tasks running in kernel space may have the same map (the kernel map). No two kernel-external tasks have the same value for $\underline{t}ask\_map$. The set $\underline{m}ap\_exists$ denotes the existing VM maps. The expression $\underline{v}m\_entries\_head(map)$ denotes the first VM_ENTRY, if one exists, in the sequence of entries associated with map.

$$
\begin{array}{|l}
\hline
\textit{VmMapStructure} \\
\textit{TaskExist} \\
\underline{t}\textit{ask\_map} : \textit{TASK} \nrightarrow \textit{VM\_MAP} \\
\underline{m}\textit{ap\_exists} : \mathbb{P}\ \textit{VM\_MAP} \\
\underline{v}\textit{m\_entries\_head} : \textit{VM\_MAP} \nrightarrow \textit{VM\_ENTRY} \\
\hline
\mathrm{dom}\ \underline{t}\textit{ask\_map} = \underline{t}\textit{ask\_exists} \\
\mathrm{dom}\ \underline{v}\textit{m\_entries\_head} \subseteq \underline{m}\textit{ap\_exists} \\
\hline
\end{array}
$$

The set $\underline{v}m\_entry\_exists$ denotes the existing VM_ENTRY elements, and the set $\underline{v}m\_entry\_submap\_p$ denotes the entries that are submaps.[20] The following functions are defined on VM_ENTRY:

- $\underline{n}ext\_vm\_entry(e)$ — denotes the next entry after e, if there is one, in the sequence of VM entries associated with some VM map,
- $\underline{v}m\_entry\_start(e)$ — denotes the starting address of e,
- $\underline{v}m\_entry\_end(e)$ — denotes the first address after the end of e,
- $\underline{v}m\_entry\_map\_object(e)$ — denotes the VM_MAP_OBJECT associated with e,
- $\underline{v}m\_entry\_offset(e)$ — denotes the offset at which e is mapped into a memory object,
- $\underline{v}m\_entry\_prot(e)$ — denotes the current protections associated with e,
- $\underline{v}m\_entry\_max\_prot(e)$ — denotes the maximum protections that e may take,
- $\underline{v}m\_entry\_inh(e)$ — denotes the inheritance option in effect for e,
- $\underline{v}m\_entry\_wire\_count(e)$ — denotes the number of times that e has been wired, and
- $\underline{v}m\_entry\_sid(e)$ — denotes the OSI associated with e.

Every existing entry has a start and an end address, an offset, a protection, a maximum protection, an inheritance option and a wire count. Every entry that is a submap has an associated map object while an entry that is not a submap might not have any associated VM_MAP_OBJECT. For convenience we define $map\_entries(map)$ to denote the set of VM_ENTRY contained in map.

[20] The only map known to have submaps is `kernel_map`. It has the following submaps:

- `device_io_map`,
- `ipc_kernel_map`,
- `kalloc_map`,
- `zone_map`, and
- the map of any task running in kernel space that does not use the entire kernel map.

$$
\begin{array}{l}
\textbf{VmEntry} \\
\hline
VmMapStructure \\
\underline{v}m\_entry\_exists : \mathbb{P}\ VM\_ENTRY \\
\underline{v}m\_entry\_submap\_p : \mathbb{P}\ VM\_ENTRY \\
\underline{n}ext\_vm\_entry : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ VM\_ENTRY \\
\underline{v}m\_entry\_start : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ PAGE\_INDEX \\
\underline{v}m\_entry\_end : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ PAGE\_INDEX \\
\underline{v}m\_entry\_map\_object : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ VM\_MAP\_OBJECT \\
\underline{v}m\_entry\_offset : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ OFFSET \\
\underline{v}m\_entry\_prot : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ \mathbb{P}\ PROTECTION \\
\underline{v}m\_entry\_max\_prot : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ \mathbb{P}\ PROTECTION \\
\underline{v}m\_entry\_inh : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ INHERITANCE\_OPTION \\
\underline{v}m\_entry\_wire\_count : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ \mathbb{N} \\
\underline{v}m\_entry\_sid : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ OSI \\
map\_entries : VM\_MAP \longrightarrow \mathbb{P}\ VM\_ENTRY \\
\hline
\mathrm{dom}\,\underline{v}m\_entry\_start = \mathrm{dom}\,\underline{v}m\_entry\_end = \mathrm{dom}\,\underline{v}m\_entry\_offset \\
\quad = \mathrm{dom}\,\underline{v}m\_entry\_prot = \mathrm{dom}\,\underline{v}m\_entry\_max\_prot = \mathrm{dom}\,\underline{v}m\_entry\_inh \\
\quad = \mathrm{dom}\,\underline{v}m\_entry\_wire\_count = \mathrm{dom}\,\underline{v}m\_entry\_sid = \underline{v}m\_entry\_exists \\
\mathrm{dom}\,\underline{n}ext\_vm\_entry \subseteq \underline{v}m\_entry\_exists \\
\underline{v}m\_entry\_submap\_p \subseteq \mathrm{dom}\,\underline{v}m\_entry\_map\_object \subseteq \underline{v}m\_entry\_exists \\
map\_entries = Gen\_set(\underline{v}m\_entries\_head, \underline{n}ext\_vm\_entry)
\end{array}
$$

The set $\underline{v}m\_map\_object\_exists$ denotes the existing VM_MAP_OBJECT elements. A VM_MAP_OBJECT may be either a memory object or another VM map. The expressions $\underline{m}ap\_object\_as\_memory(obj)$ and $\underline{m}ap\_object\_as\_submap(obj)$ denote the associated memory object or VM map. The domains of these two functions partition the set of existing map objects. The function $\underline{v}m\_entry\_map\_object$ maps a submap entry to an element of the domain of $\underline{m}ap\_object\_as\_submap$, and it maps other entries to elements of the domain of $\underline{m}ap\_object\_as\_memory$. For convenience, we define the functions vm_entry_memory and vm_entry_submap as the compositions of $\underline{v}m\_entry\_map\_object$ with $\underline{m}ap\_object\_as\_memory$ and $\underline{m}ap\_object\_as\_submap$, respectively.

$$
\begin{array}{l}
\textbf{VmMapObject} \\
\hline
VmEntry \\
\underline{v}m\_map\_object\_exists : \mathbb{P}\ VM\_MAP\_OBJECT \\
\underline{m}ap\_object\_as\_memory : VM\_MAP\_OBJECT \rightarrowtail\!\!\!\!\!+ MEMORY \\
\underline{m}ap\_object\_as\_submap : VM\_MAP\_OBJECT \rightarrowtail\!\!\!\!\!+ VM\_MAP \\
vm\_entry\_memory : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ MEMORY \\
vm\_entry\_submap : VM\_ENTRY \leftrightarrow\!\!\!\!\!+ VM\_MAP \\
\hline
\langle \mathrm{dom}\,\underline{m}ap\_object\_as\_memory, \mathrm{dom}\,\underline{m}ap\_object\_as\_submap \rangle \\
\quad \mathsf{partition}\ \underline{v}m\_map\_object\_exists \\
\mathrm{dom}\,\underline{m}ap\_object\_as\_memory = \mathrm{ran}(\underline{v}m\_entry\_submap\_p \ntriangleleft \underline{v}m\_entry\_map\_object) \\
\mathrm{dom}\,\underline{m}ap\_object\_as\_submap = \mathrm{ran}(\underline{v}m\_entry\_submap\_p \lhd \underline{v}m\_entry\_map\_object) \\
vm\_entry\_memory = \underline{v}m\_entry\_map\_object \mathbin{⨾} \underline{m}ap\_object\_as\_memory \\
vm\_entry\_submap = \underline{v}m\_entry\_map\_object \mathbin{⨾} \underline{m}ap\_object\_as\_submap
\end{array}
$$

We define a global function $Page\_index\_int$ that maps a $PAGE\_INDEX$ to a non-negative integer. This allows the numeric comparison of page addresses.

$$
Page\_index\_int : PAGE\_INDEX \rightarrowtail \mathbb{N}
$$

For use in refining the model of the VM system, we define the functions vm_map_lookup and $vm\_map\_lookup\_entry$ which each map a $(tk, pindex)$ pair to a VM_ENTRY. For vm_map_lookup, the pair is mapped to $entry$ if and only if $entry$ is not a submap, and there is a non-empty sequence $lookup\_seq$ of $VM\_MAP \times VM\_ENTRY$ pairs such that

- $\underline{t}ask\_map(tk)$ is the first component of the first element of $lookup\_seq$,
- $entry$ is the second component of the last element of $lookup\_seq$,
- for each element (m, e) of $lookup\_seq$:
  - e is in the set of entries for m,
  - $pindex$ is in the address range defined by the start and end addresses of e,
  - if (m, e) is not the last element of $lookup\_seq$, then e is a submap entry with the first component of the next pair in the sequence as its submap.

For $vm\_map\_lookup\_entry$, a $(tk, pindex)$ pair is mapped to $entry$ if and only if $entry$ is in the set of entries for $\underline{t}ask\_map(tk)$, and $pindex$ is in the address range defined by the start and end addresses of $entry$,

$$
\begin{array}{l}
\textbf{VmLookup} \\
\hline
VmMapStructure \\
VmEntry \\
VmMapObject \\
vm\_map\_lookup\_entry : TASK \times PAGE\_INDEX \nrightarrow VM\_ENTRY \\
vm\_map\_lookup : TASK \times PAGE\_INDEX \nrightarrow VM\_ENTRY \\
\hline
\forall\, entry : VM\_ENTRY;\ tk : TASK;\ pindex : PAGE\_INDEX \\
\bullet\ entry = vm\_map\_lookup\_entry(tk, pindex) \\
\Leftrightarrow (\exists\, map : VM\_MAP \\
\quad \bullet\ \underline{t}ask\_map(tk) = map \\
\quad \land\ entry \in map\_entries(map) \\
\quad \land\ Page\_index\_int(\underline{v}m\_entry\_start(entry)) \leq Page\_index\_int(pindex) \\
\qquad < Page\_index\_int(\underline{v}m\_entry\_end(entry))) \\
\forall\, entry : VM\_ENTRY;\ tk : TASK;\ pindex : PAGE\_INDEX \\
\bullet\ entry = vm\_map\_lookup(tk, pindex) \\
\Leftrightarrow (entry \notin \underline{v}m\_entry\_submap\_p \\
\quad \land\ (\exists\, lookup\_seq : \mathrm{seq}_1(VM\_MAP \times VM\_ENTRY) \\
\qquad \bullet\ \underline{t}ask\_map(tk) = first(head\ lookup\_seq) \\
\qquad \land\ entry = second(last\ lookup\_seq) \\
\qquad \land\ (\forall\, i : 1 \mathrel{..} \#lookup\_seq;\ e : VM\_ENTRY \\
\qquad\quad \mid e = second(lookup\_seq(i)) \\
\qquad\quad \bullet\ e \in map\_entries(first(lookup\_seq(i))) \\
\qquad\quad \land\ Page\_index\_int(\underline{v}m\_entry\_start(e)) \leq Page\_index\_int(pindex) \\
\qquad\qquad < Page\_index\_int(\underline{v}m\_entry\_end(e)) \\
\qquad\quad \land\ (i < \#lookup\_seq \\
\qquad\qquad \Rightarrow (e, first(lookup\_seq(i+1))) \in vm\_entry\_submap))))
\end{array}
$$

Now, we define $mapped\_memory$ and $mapped\_offset$ by composing vm_map_lookup with vm_entry_memory and $\underline{v}m\_entry\_offset$, respectively. We define $mapped\_offset$, $protection$, $\underline{m}ax\_protection$, $\underline{i}nheritance$ and $\underline{w}ire\_count$ by composing $vm\_map\_lookup\_entry$ with the appropriate VM entry functions.

Editorial Note:
It is unclear whether $mapped\_memory$ and $mapped\_offset$ are best defined as below or whether it would be better to use $vm\_map\_lookup\_entry$ for them as well. The question is whether, when $pindex$ denotes a submap for task tk, the pair $(tk, pindex)$ should be in the domains of $mapped\_memory$ and $mapped\_offset$. The prototype appears to follow the submap link when dealing with page faults. However, when accessing and returning state information associated with a region, it does not look at the submap. Furthermore, **vm_region** returns a null name for the memory object when the address leads to a submap.

VmRefinement

$VmLookup$
$AddressSpace$
$Protection$
$Inheritance$
$Wired$
$PageSid$

$mapped\_memory = vm\_map\_lookup \mathbin{;} vm\_entry\_memory$
$mapped\_offset = vm\_map\_lookup \mathbin{;} \underline{v}m\_entry\_offset$
$protection = vm\_map\_lookup\_entry \mathbin{;} \underline{v}m\_entry\_prot$
$\underline{m}ax\_protection = vm\_map\_lookup\_entry \mathbin{;} \underline{v}m\_entry\_max\_prot$
$\underline{i}nheritance = vm\_map\_lookup\_entry \mathbin{;} \underline{v}m\_entry\_inh$
$\underline{w}ire\_count = vm\_map\_lookup\_entry \mathbin{;} \underline{v}m\_entry\_wire\_count$
$\underline{p}age\_sid = vm\_map\_lookup\_entry \mathbin{;} \underline{v}m\_entry\_sid$

D.5 Miscellaneous Refinements

The expression $\underline{t}hreads\_head(tk)$ denotes the first $THREAD$, if one exists, in the sequence of threads belonging to task tk. The expression $\underline{n}ext\_thread(th)$ denotes the successor $THREAD$ of th, if one exists, in the sequence of threads belonging to the owning task of thread th.

ThreadList

$TasksAndThreads$
$\underline{t}hreads\_head : TASK \pfun THREAD$
$\underline{n}ext\_thread : THREAD \pfun THREAD$

$threads = Gen\_set(\underline{t}hreads\_head, \underline{n}ext\_thread)$

The expression $\underline{p}rocessors\_head(pset)$ denotes the first $PROCESSOR$, if one exists, in the sequence of processors belonging to processor set $pset$. The expression $\underline{n}ext\_processor(proc)$ denotes the successor $PROCESSOR$ of $proc$, if one exists, in the sequence of processors belonging to the processor set of which $proc$ is a member.

ProcessorList

$ProcessorAndProcessorSet$
$\underline{p}rocessors\_head : PROCESSOR\_SET \pfun PROCESSOR$
$\underline{n}ext\_processor : PROCESSOR \pfun PROCESSOR$

$processors = Gen\_set(\underline{p}rocessors\_head, \underline{n}ext\_processor)$

The expression $\underline{a}ssigned\_tasks\_head(pset)$ denotes the first $TASK$, if one exists, in the sequence of tasks belonging to processor set $pset$. The expression $\underline{n}ext\_assigned\_task(tk)$ denotes the successor $TASK$ of tk, if one exists, in the sequence of tasks belonging to the processor set to which tk is assigned.

$AssignedTaskList$

$$TaskAndProcessorSet$$
$$\underline{a}ssigned\_tasks\_head : PROCESSOR\_SET \nrightarrow TASK$$
$$\underline{n}ext\_assigned\_task : TASK \nrightarrow TASK$$

$$have\_assigned\_tasks = Gen\_set(\underline{a}ssigned\_tasks\_head, \underline{n}ext\_assigned\_task)$$

The expression $\underline{a}ssigned\_threads\_head(pset)$ denotes the first $THREAD$, if one exists, in the sequence of threads belonging to processor set $pset$. The expression $\underline{n}ext\_assigned\_thread(th)$ denotes the successor $THREAD$ of th, if one exists, in the sequence of threads belonging to the processor set to which th is assigned.

$AssignedThreadList$

$$ThreadAndProcessorSet$$
$$\underline{a}ssigned\_threads\_head : PROCESSOR\_SET \nrightarrow THREAD$$
$$\underline{n}ext\_assigned\_thread : THREAD \nrightarrow THREAD$$

$$have\_assigned\_threads = Gen\_set(\underline{a}ssigned\_threads\_head, \underline{n}ext\_assigned\_thread)$$

The expression $\underline{m}essages\_head(port)$ denotes the first $MESSAGE$, if one exists, in the sequence of messages waiting in $port$. The expression $\underline{n}ext\_message(msg)$ denotes the successor $MESSAGE$ of msg, if one exists, in the sequence of messages waiting in the port in which msg is waiting.

Editorial Note:
This says nothing about the messages in a queue of a port set. This queue is not currently modeled, so there is nothing to refine. If we add port set message queues, the refinement would appear nearly identical to the following refinement.

$MessageInPortList$

$$MessageQueues$$
$$\underline{m}essages\_head : PORT \nrightarrow MESSAGE$$
$$\underline{n}ext\_message : MESSAGE \nrightarrow MESSAGE$$

$$\underline{m}essage\_in\_port\_rel = Gen\_seq(\underline{m}essages\_head, \underline{n}ext\_message)$$

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols

M

S