A Mathematical Model of the Mach Kernel: Kernel Requests

William R. Bevier and Lawrence M. Smith

Technical Report 103 December, 1994

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

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

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

Contents

Chapter 1

Introduction

This report presents a partial specification for Mach kernel requests. It is based on a mathematical description of a legal Mach kernel state given in [BS94b]. That report gives a specification of the entities that may exist in a legal Mach kernel state, and their properties. We restrict our attention in this report to those requests that can be described in terms that are axiomatized in [BS94b]. Therefore, we describe only a strict subset of the requests discussed in the Mach kernel interface manual [Loe91]. An overview of the legal state specification and kernel request specification can be found in [BS94a]. In particular, it provides a detailed introduction to specifications for kernel requests. Familiarity with [BS94a] and [BS94b] is assumed and is important.

The actions required of a kernel request are specified by the assertion and disassertion of relations on entities. If one understands the abstract relations in which entities can participate, one can follow this specification entirely at that level. The ability to do so is one of the main benefits of a specification. To understand how the specification applies to a Mach implementation, one must understand how these relations are implemented. In fact, the implementation of each relation is relatively straightforward in the Mach 3.0 kernel. Notes on how how entities and relations are implemented in Mach 3.0 are given in [BS94b].

This work was not done in collaboration of Mach 3.0 implementors. For this reason, as well as the absence of any means other than human inspection to verify the implementation with respect to this specification, this document may not reflect aspects of the latest design. Additionally, the authors may have made some mistakes. We have tried our best to accurately reflect a large part of the Mach 3.0 design. We believe that we have addressed the most stable design elements.

The format of a kernel request specification is similar to the documentation for a request in a kernel interface manual. We summarize functionality, and describe input parameters and returned values. Subsequently, we give formulas which state requirements on computations which are legal implementations of the request. Formulas are augmented with English text.

One of the most important sections in this document is Chapter 7. Each kernel request specification appeals to primitive relations introduced originally in [BS94b], and to common, intermediate specifications defined in terms of those primitives. These common definitions are collected in Section 7, and provide detailed requirements on the behavior of a kernel request that are difficult to make both concise and precise in English.

The specification for `mach_msg` occurs first in this document, mimicking the organization of [Loe91]. However, this is one of the most complex interfaces, and we suggest that the reader look first at a simpler specification, such as `task_create`, to get a feel for the notation.

Notation

A detailed discussion of notation is given in [BS94a]. We present a summary of those conventions, and a few additional conventions followed in this report.

Fonts

We use font shifts to provide visual clues for the uses of identifier and function names in formulas.

- "**var**" – A valuable printed bold is a free variable in a temporal formula. It represents an input parameter or output variable.

- "*var*" – A variable printed in italics is one that is bound in the formula by a quantifier.

- "`'flag`" – An identifier in typewriter font preceded by an apostrophe is a literal. These are used to represent constants like return code values.

- "NULLNAME" – Small capitals also denote constants, but are typically used to identify system constants like the value of a null port name.

- "local-namep (...)" – A function whose name is in lower-case roman is a primitive state term, introduced in the legal state specification.

- "Mach-Port-Move-Member-Invalid-Name (...)" – A function whose name is capitalized is a recognizer for a temporal behavior[1].

Temporal Logic

A kernel request specification is a predicate on a *behavior* — a sequence of kernel states in which actions among various agents are interleaved. A kernel request implementation satisfies the specification if the predicate holds on all behaviors that it can generate.

The temporal notation can indicate that an action is taken by an agent — the entity in behalf of whom the actions take place. Steps involved in a single thread of kernel behavior, and only those steps, are labeled with the same agent. One can think of the agent as a stack of thread identifiers constructed by remote procedure calls. The topmost element of the stack is the identifier of the thread executing in behalf of the next thread on the stack, and so on.

Parameters to kernel request specifications correspond to parameters to the actual kernel requests. In the Mach 3.0 user interface, the actual parameter supplied for some entity is a port which represents that entity. A simple computation finds the entity represented by the port. This specification ignores that interface. Parameters to the specification are members of the intended entity class, not a port which represents a member of the class.

[1]If the argument list is empty, the parentheses are omitted.

The agent of a kernel request is thought of as an input parameter. However, in these specifications, for the sake of economy, we do not explicitly declare the agent as an input. There is always one agent of interest, which we write as α.

Similarly, the return code is an out parameter of every request. We denote the return code parameter by the symbol **rc**. We omit this declaration from the list of parameters, but use **rc** in the specifications.

Here are examples of the temporal logic notation we use in writing specifications. See [BS94a] for a thorough discussion.

$\Diamond \mathrm{taskp}\,(\mathbf{t})[\alpha]$	Eventually. Agent α reaches a state in which $\mathbf{t}$ is a task.
$\Diamond{\uparrow}\mathrm{taskp}\,(\mathbf{t})[\alpha]$	Eventually asserted. Agent α "creates" task $\mathbf{t}$. In other words, the state predicate taskp($\mathbf{t}$) is asserted in an α-step.
$\Diamond{\downarrow}\mathrm{taskp}\,(\mathbf{t})[\alpha]$	Eventually disasserted. Agent α "destroys" task $\mathbf{t}$. In other words, the state predicate taskp($\mathbf{t}$) is disasserted in an α-step.
$\Diamond{\dagger}\mathrm{taskp}\,(\mathbf{t})[\alpha]$	Eventual interference. Some agent other than α eventually modifies property taskp($\mathbf{t}$). This is usually an error condition.
$\Diamond p \ ; \Diamond q$	Sequential Composition. Property p eventually occurs, followed eventually by property q.
$p \wedge q$	Conjunction. (p and q).
$p \vee q$	Disjunction. (p or q).
$\neg\, p$	Negation. (not p).
$\forall\, 0 \leq x < n\colon\ \mathrm{p}\,(x)$	Universal quantification.
$\exists\, 0 \leq x < n\colon\ \mathrm{p}\,(x)$	Existential quantification.

Chapter 2

IPC Interface

2.1 Introduction

Inter-process communication is accomplished in Mach with the `mach_msg` kernel service. This service allows a caller to send a message, receive a message, or both. The latter case, the most general, is used in a remote procedure call.

A remote procedure call from a client task to a server task involves a number of steps.

1. The client task sets up a message buffer data structure in its address space. The contents of the message buffer encode instructions from the task to the kernel for how the message to the server should be constructed. It also specifies the side effects the task would like to see in its own resources.

2. The client task invokes `mach_msg` requesting a send followed by a receive. The client task gives to the kernel its local names for three ports: the destination of the sent message, the reply port (to which a the client requests a response to the message from the server), and the source for the received message. The last two are typically the same.

3. The kernel interprets the contents of the message buffer, constructing the message from the client task's resources.

4. The kernel queues the message in the port.
5. The kernel processes the receive half of the client task's request. Typically, the receive port's queue will be empty, so it waits.
6. At a later time, the server requests that a message be dequeued.
7. The kernel fills a message buffer in the server task that describes the message and the side effects it caused in the server's resources. The side effects include inserting a send right to the reply port.
8. The server uses the same mechanism to return the response to the sender. (Typically it will receive its next request with the same invocation of `mach_msg`. In this case, no reply port is specified.)
9. The original client task completes its receive.

Our specification for `mach_msg` corresponds to this decomposition of steps.

In Section 2.2 we introduce the high-level specification of `mach_msg`. The high-level specification references predicates describing the sending and receiving halves of the computation.

Sections 2.3 through 2.6 describe the sending and receiving operations in success and failure cases.

In Section 2.7 we introduce the *message descriptor*, which is an abstraction of Mach's message buffer data structure. The message descriptor is a specification artifact that allows us to ignore the details of the layout of the encoding in the task's address space. A part of sending a message is creating the message from the resources of a task according to the instructions in a message descriptor. This operation is specified in Section 2.8.

Receiving a message causes rights and data to become a part of the receiving task's resources, and a message descriptor describes the transition. This operation is specified in Section 2.9. [1]

[1] The same mechanism is used on a send operation when it times out or is interrupted. This "pseudo-receive" operation readies the message buffer and its contents for a retry. We do not specify this level of detail.

2.2 mach_msg

DESCRIPTION

Send and/or receive a message.

PARAMETERS

t. The invoking task entity.

options. A set of tokens from {`'send`, `'receive`, `'send-timeout`, `'receive-timeout`}.

send-name. If `'send` $\in$**options**, the local name of the port to which a message will be sent.

send-instr. Instruction for disposition of the send port name, one of `'make`, `'copy`, or `'move`.

send-type. The type of right to be provided for the send port, one of `'send` or `'send-once`.

rcv-name. If `'receive` $\in$**options**, the local name for the receive right of the port or port set from which a message will be received.

reply-name. For send, the local name of the port which will be the reply port of the sent message, if any. For receive, the local name for the reply port in the message. If no reply port is desired, **reply-name** =NULLNAME, **reply-instr** =`'none`, and **reply-type** =`'none`.

reply-instr. Instruction for disposition of the send reply port name, one of `'make`, `'copy`, `'move`, or `'none`.

reply-type. For send, the type of right to be provided for the send port, one of `'send`, `'send-once`, or `'none`. For receive, the right type for the *reply-name*, either `'send` or `'send-once`.

md. A message descriptor. For send, the message descriptor is provided by the calling task describing the contents of the message to send. For receive, it is constructed by the kernel to describe the message to the caller. The message descriptor is a specification artifact abstracting the encoding of instructions in the portion of the task's address space designated as the message buffer.

OUTCOMES

The Mach-Msgp predicate is specified to recognize one of five cases.

1. Neither `'send` nor `'receive` are in **options**, so `mach_msg` is a no-op.
2. The flag `'send` is in **options**, but not `'receive`.
3. The flag `'receive` is in **options**, but not `'send`.
4. Both `'send` and `'receive` are in **options**, but the send fails. The receive is not attempted.
5. Both `'send` and `'receive` are in **options**, and the send succeeds.

The cases decompose using four more primitive predicates, describing send success, send failure, receive success, and receive failure.

$$\begin{array}{ll} & \text{Mach-Msgp} \\ \equiv & \quad \text{Mach-Msg-Noop} \\ & \vee \ \text{Mach-Msg-Sendp} \\ & \vee \ \text{Mach-Msg-Rcvp} \\ & \vee \ \text{Mach-Msg-Send-Failure-No-Rcvp} \\ & \vee \ \text{Mach-Msg-Send-Success-Rcvp} \end{array}$$

$$\begin{array}{ll} & \text{Mach-Msg-Noop} \\ \equiv & \quad (\texttt{'send} \notin \mathbf{options})[\alpha] \\ & \wedge \ (\texttt{'receive} \notin \mathbf{options})[\alpha] \\ & \wedge \ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-msg-success})[\alpha] \end{array}$$

$$\begin{array}{ll} & \text{Mach-Msg-Sendp} \\ \equiv & \quad (\texttt{'send} \in \mathbf{options})[\alpha] \\ & \wedge \ (\texttt{'receive} \notin \mathbf{options})[\alpha] \\ & \wedge \ (\text{Mach-Msg-Send-Successp} \vee \text{Mach-Msg-Send-Failurep}) \end{array}$$

$$
\begin{aligned}
&\text{Mach-Msg-Rcvp}\\
\equiv\ &\quad (\texttt{'send} \in \mathbf{options})[\alpha]\\
&\wedge\ (\texttt{'receive} \notin \mathbf{options})[\alpha]\\
&\wedge\ (\text{Mach-Msg-Rcv-Successp} \vee \text{Mach-Msg-Rcv-Failurep})
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Msg-Send-Failure-No-Rcvp}\\
\equiv\ &\quad (\texttt{'send} \in \mathbf{options})[\alpha]\\
&\wedge\ (\texttt{'receive} \in \mathbf{options})[\alpha]\\
&\wedge\ \text{Mach-Msg-Send-Failurep}
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Msg-Send-Success-Rcvp}\\
\equiv\ &\quad (\texttt{'send} \in \mathbf{options})[\alpha]\\
&\wedge\ (\texttt{'receive} \in \mathbf{options})[\alpha]\\
&\wedge\ (\text{Mach-Msg-Send-Successp}\ ;\ \text{Mach-Msg-Rcvp})
\end{aligned}
$$

2.3 Send Success

A computation is recognized as an instance of Mach-Msg-Send-Successp if

- the local names **send-name** and **reply-name** are found to name proper rights to destination and reply ports entities,
- a message is constructed from the message descriptor,
- the reply port (if any) is associated with the message, and
- the message is enqueued on the destination port.

$$
\begin{aligned}
&\text{Mach-Msg-Send-Successp}\\
\equiv\ &\exists\ \textit{send-port} \in \text{ALL-ENTITIES},\\
&\quad \textit{reply-port} \in (\text{ALL-ENTITIES} \cup \{\text{NULL-PTR}\}),\ \textit{mg} \in \text{ALL-ENTITIES}:\\
&(\quad \Diamond(\quad \text{Extract-Destination}\,(\mathbf{send\text{-}name},\ \mathbf{send\text{-}instr},\\
&\qquad\qquad\qquad\qquad \mathbf{send\text{-}type},\ \textit{send-port})\\
&\qquad\quad \wedge\ \text{Extract-Reply}\,(\mathbf{reply\text{-}name},\ \mathbf{reply\text{-}instr},\\
&\qquad\qquad\qquad\qquad \mathbf{reply\text{-}type},\ \textit{reply-port})\\
&\qquad\quad \wedge\ \uparrow\!\text{messagep}\,(\textit{mg})[\alpha])\\
&\quad ;\ \text{Message-Is-Constructed}\,(\textit{mg},\ \textit{reply-port})\\
&\quad ;\ \text{Message-Is-Queuedp}\,(\textit{mg},\ \textit{send-port})\\
&\quad ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-msg-success})[\alpha])
\end{aligned}
$$

$$\begin{array}{ll} & \text{Message-Is-Constructed}\,(mg,\ \textit{reply-port}) \\ \equiv & \quad \text{Md-To-Messagep}\,(\mathbf{md},\ |\mathbf{md}|,\ mg) \\ & \wedge\ (\quad (\textit{reply-port} \neq \textsc{null-ptr})[\alpha] \\ & \qquad \rightarrow \Diamond\!\uparrow\!\text{reply-port-rel}\,(mg,\ \textit{reply-port},\ \textbf{reply-type})[\alpha]) \end{array}$$

$$\begin{array}{ll} & \text{Message-Is-Queuedp}\,(mg,\ p) \\ \equiv & \exists\ \textit{queue} = \text{messages}\,(p)\text{:}\ \ (\Diamond\!\uparrow\!(\text{messages}\,(p) = \text{append}\,(\textit{queue},\ \langle mg \rangle))[\alpha]) \end{array}$$

2.4 Send Failure

A send operation can fail for a number of reasons.

$$\begin{array}{ll} & \text{Mach-Msg-Send-Failurep} \\ \equiv & \quad \text{Mach-Msg-Send-Header-Error} \\ & \vee\ \text{Mach-Msg-Md-Error} \\ & \vee\ \text{Mach-Msg-Bad-Message-Body} \\ & \vee\ \text{Mach-Msg-Send-Timeout} \\ & \vee\ \text{Mach-Msg-Send-Interrupted} \\ & \vee\ \text{Mach-Msg-Send-Resource-Shortage} \end{array}$$

Problems with header arguments

This section contains descriptions of return codes which result from problems with arguments. The *invalid destination* return code results when the destination (send) port is invalid. The *invalid reply* outcome results when there is a problem determining the reply port from the arguments. The *invalid header* outcome results when a field in the header had a bad value.

$$\begin{array}{ll} & \text{Mach-Msg-Send-Header-Error} \\ \equiv & \quad \text{Mach-Send-Invalid-Dest-Portp} \\ & \vee\ \text{Mach-Send-Invalid-Reply-Portp} \\ & \vee\ \text{Mach-Send-Invalid-Header-Types} \end{array}$$

$$\begin{array}{ll} & \text{Mach-Send-Invalid-Header-Types} \\ \equiv & \quad (\quad (\textbf{send-type} = \texttt{'receive})[\alpha] \\ & \qquad \vee\ (\textbf{reply-type} = \texttt{'receive})[\alpha]) \\ & \wedge\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-send-invalid-header})[\alpha] \end{array}$$

$$\begin{array}{l}
\quad \text{Mach-Send-Invalid-Reply-Portp} \\
\equiv \Diamond(\quad (\textbf{reply-name} \neq \textsc{nullname})[\alpha] \\
\qquad \wedge\ \Diamond(\quad \neg\ \text{Legal-Name-Instr-Right}\,(\mathbf{t},\ \textbf{reply-name}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{reply-instr},\ \textbf{reply-type}) \\
\qquad\qquad \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-send-invalid-reply})[\alpha]))
\end{array}$$

$$\begin{array}{l}
\quad \text{Mach-Send-Invalid-Dest-Portp} \\
\equiv \Diamond(\quad \neg\ \text{Legal-Name-Instr-Right}\,(\mathbf{t},\ \textbf{send-name},\ \textbf{send-instr}, \\
\qquad\qquad\qquad\qquad\qquad \textbf{send-type}) \\
\qquad \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-send-invalid-dest})[\alpha])
\end{array}$$

Problems with the message body

This set of return codes indicate that a bad message descriptor element was discovered. *Invalid Right* occurs when a port right in a message descriptor element is not valid. *Invalid Data* occurs when a region of memory designated as an out-of-line message element is either unallocated or read-protected.

$$\begin{array}{l}
\quad \text{Mach-Msg-Bad-Message-Body} \\
\equiv \text{Md-Exists-Invalid-Right} \vee \text{Md-Exists-Invalid-Memory}
\end{array}$$

$$\begin{array}{l}
\quad \text{Md-Exists-Invalid-Right} \\
\equiv \exists\ 0 \leq i < |\mathbf{md}|,\ n \in \mathcal{N},\ r \in \mathcal{R},\ instr \in \{\texttt{'make},\ \texttt{'copy},\ \texttt{'move}\}: \\
\quad (\Diamond(\quad (\mathbf{md}_i = \langle \texttt{'right},\ n,\ instr,\ r\rangle)[\alpha] \\
\qquad \wedge\ \neg\ \text{Legal-Name-Instr-Right-Deadok}\,(\mathbf{t},\ n,\ instr,\ r) \\
\qquad \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-send-invalid-right})[\alpha]))
\end{array}$$

$$\begin{array}{l}
\quad \text{Md-Exists-Invalid-Memory} \\
\equiv \exists\ 0 \leq i < |\mathbf{md}|,\ 0 \leq va < \textsc{address-space-limit}, \\
\qquad 0 \leq l < \textsc{address-space-limit},\ instr \in \{\texttt{'delete},\ \texttt{'no-delete}\}, \\
\qquad va \leq va1 < (va + l): \\
\quad (\Diamond(\quad (\mathbf{md}_i = \langle \texttt{'data},\ \texttt{'out-of-line},\ va,\ l,\ instr\rangle)[\alpha] \\
\qquad \wedge\ (\quad (\neg\ \text{allocated}\,(\mathbf{t},\ va1))[\alpha] \\
\qquad\quad \vee \quad \text{allocated}\,(\mathbf{t},\ va1)[\alpha] \\
\qquad\qquad \wedge\ (\texttt{'read} \notin \text{protection}\,(\mathbf{t},\ \text{trunc-page}\,(va1)))[\alpha]) \\
\qquad \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-send-invalid-memory})[\alpha]))
\end{array}$$

Other Send Failures

The send operation will return a distinguished error code if the send blocks, then either times out or is interrupted.

The kernel returns an error if the message buffer is badly formatted[2].

$$\begin{aligned}&\text{Mach-Msg-Md-Error}\\ \equiv\ & \neg\ \mathrm{Mdp}\,(\mathbf{md}) \wedge \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-msg-bad-message})[\alpha]\end{aligned}$$

$$\text{Mach-Msg-Send-Interrupted} \equiv \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-send-interrupted})[\alpha]$$

$$\begin{aligned}&\text{Mach-Msg-Send-Timeout}\\ \equiv\ & \quad (\texttt{'send-timeout} \in \mathbf{options})[\alpha]\\ & \wedge\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-send-timed-out})[\alpha]\end{aligned}$$

There are a number of ways a send can fail due to resource shortages.

$$\begin{aligned}&\text{Mach-Msg-Send-Resource-Shortage}\\ \equiv\ & \quad \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-msg-ipc-space-send})[\alpha]\\ & \vee\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-msg-vm-space-send})[\alpha]\\ & \vee\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-msg-ipc-kernel-send})[\alpha]\\ & \vee\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-msg-vm-kernel-send})[\alpha]\\ & \vee\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'mach-send-no-buffer})[\alpha]\end{aligned}$$

2.5 Receive Success

When `mach_msg` is successful, the message is dequeued, the reply port (if any) is taken from the message, and the contents of the message are transferred to the receiver.

Receiving a message has four parts.

- The receiving thread blocks until the message queue is not empty. This part does not appear in our specification. We just say that the message is eventually received (or not).

- The message is dequeued from the port.

[2]In the implementation of Mach 3.0, `'mach-msg-bad-message` is refined to either `'mach-send-msg-too-small` or `'mach-send-invalid-data`.

- A right to the reply port (if any) is taken from the message, and the contents of the message are are extracted from the message and inserted into the task's resources.

- The message descriptor describing the message is created.

$$
\begin{array}{ll}
 & \text{Mach-Msg-Rcv-Successp} \\
\equiv & \exists\ \textit{rcv-port} \in \text{ALL-ENTITIES},\ \textit{reply-port} \in \text{ALL-ENTITIES}, \\
 & \quad \textit{mg} \in \text{ALL-ENTITIES}: \\
 & \quad (\quad \Diamond\text{Names-Receiving-Port}\,(\mathbf{rcv\text{-}name},\ \textit{rcv-port}) \\
 & \quad\ ;\ \text{Message-Dequeuedp}\,(\textit{mg},\ \textit{rcv-port}) \\
 & \quad\ ;\ \text{Port-Is-Message-Reply-Port}\,(\textit{reply-port},\ \textit{mg}) \\
 & \quad\ ;\ \text{Message-To-Mdp}\,(\textit{mg},\ \mathbf{md}) \\
 & \quad\ ;\ \downarrow\text{entityp}\,(\textit{mg})[\alpha] \\
 & \quad\ ;\ \text{Insert-Reply-Port}\,(\mathbf{reply\text{-}name},\ \mathbf{reply\text{-}type},\ \textit{reply-port}) \\
 & \quad\ ;\ \uparrow(\mathbf{rc} = \texttt{'mach-msg-success})[\alpha])
\end{array}
$$

The predicate Message-Dequeuedp recognizes a computation where the message is the first in the port's queue, and all messsages are moved up one position.

$$
\begin{array}{ll}
 & \text{Message-Dequeuedp}\,(\textit{mg},\ p) \\
\equiv & \Diamond\exists\ \textit{queue} = \text{messages}\,(p): \\
 & \quad (\Diamond\uparrow(\textit{mg} = \textit{queue}_{\mathtt{0}})[\alpha]\ ;\ \Diamond\uparrow(\text{messages}\,(p) = \textit{queue}_{\mathtt{1}..|\textit{queue}|})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Port-Is-Message-Reply-Port}\,(p,\ \textit{mg}) \\
\equiv & \Diamond(\qquad \text{exists-reply-port}\,(\textit{mg})[\alpha] \\
 & \qquad\ \wedge\ \Diamond\uparrow(p = \text{reply-port}\,(\textit{mg}))[\alpha] \\
 & \qquad\ \wedge\ \Diamond\uparrow(\textit{reply-type} = \text{reply-right}\,(\textit{mg}))[\alpha] \\
 & \quad \vee\ (\neg\ \text{exists-reply-port}\,(\textit{mg}))[\alpha] \wedge \Diamond\uparrow(p = \text{NULL-PTR})[\alpha])
\end{array}
$$

2.6 Receive Failure

Receive can fail for a number of reasons.

$$\begin{array}{ll} & \text{Mach-Msg-Rcv-Failurep} \\ \equiv & \quad \text{Mach-Msg-Rcv-Header-Error} \\ & \vee\ \text{Mach-Msg-Rcv-Timeout} \\ & \vee\ \text{Mach-Msg-Rcv-Interrupted} \\ & \vee\ \text{Mach-Msg-Rcv-Resource-Shortage} \end{array}$$

2.6.1 Problems with Header Arguments

"Header errors" result from problems with the port from which the message is to be received. Either

- the local name does not designate a receive right or port set, or
- the local name is a receive right which is a member of a port set, or
- or the local name designates a receive right or port set, but loses that designation sometime during the computation, or
- the local name is a receive right that is moved into a port set at some point during the computation.

$$\begin{array}{ll} & \text{Mach-Msg-Rcv-Header-Error} \\ \equiv & \quad \text{Mach-Rcv-Invalid-Namep} \\ & \vee\ \text{Mach-Rcv-In-Port-Setp} \\ & \vee\ \text{Mach-Rcv-Port-Diedp} \\ & \vee\ \text{Mach-Rcv-Port-Changedp} \end{array}$$

$$\begin{array}{ll} & \text{Mach-Rcv-Invalid-Namep} \\ \equiv & \Diamond(\quad (\neg\ \text{r-right}\,(\mathbf{t}, \mathbf{rcv\text{-}name}))[\alpha] \\ & \quad \wedge\ \neg\ \text{port-set-namep}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] \\ & \quad \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-rcv-invalid-name})[\alpha]) \end{array}$$

$$\begin{array}{ll} & \text{Mach-Rcv-In-Port-Setp} \\ \equiv & \Diamond\exists\, n_1 \in \mathcal{N}: \\ & \quad (\quad \text{r-right}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] \\ & \quad\ \ \wedge\ \text{in-port-set}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] \\ & \quad\ \ \wedge\ \Diamond\uparrow(\mathbf{rc} = \texttt{'mach-rcv-in-set})[\alpha]) \end{array}$$

$$\begin{array}{lllll}
 & \multicolumn{4}{l}{\text{Mach-Rcv-Port-Diedp}} \\
\equiv & \Diamond(& \text{r-right}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] & & \\
 & & \wedge\ \Diamond(& \dagger\text{r-right}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] & \\
 & & & \wedge\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-rcv-port-died})[\alpha])) & \\
 \vee & \Diamond(& \text{port-set-namep}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] & & \\
 & & \wedge\ \Diamond(& \dagger\text{port-set-namep}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] & \\
 & & & \wedge\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-rcv-port-died})[\alpha]))
\end{array}$$

$$\begin{array}{ll}
 & \text{Mach-Rcv-Port-Changedp} \\
\equiv & (\quad \Diamond\text{r-right}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] \\
 & \ ;\ (\neg\ \text{in-port-set}\,(\mathbf{t}, \mathbf{rcv\text{-}name}))[\alpha] \\
 & \ ;\ \dagger\neg\ \text{in-port-set}\,(\mathbf{t}, \mathbf{rcv\text{-}name})[\alpha] \\
 & \ ;\ \uparrow\!(\mathbf{rc} = \texttt{'mach-rcv-port-changed})[\alpha])
\end{array}$$

2.6.2 Other Receive Errors

The *timed out* outcome results when a message was not received within the timeout value. The *interrupted* outcome results when the agent thread receives a software interrupt.

$$\text{Mach-Msg-Rcv-Interrupted} \equiv \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-rcv-interrupted})[\alpha]$$

$$\begin{array}{ll}
 & \text{Mach-Msg-Rcv-Timeout} \\
\equiv & \quad (\texttt{'rcv-timeout} \in \mathbf{options})[\alpha] \\
 & \wedge\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-rcv-timed-out})[\alpha]
\end{array}$$

A receive can also fail when a resource shortage occurs.

$$\begin{array}{ll}
 & \text{Mach-Msg-Rcv-Resource-Shortage} \\
\equiv & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'mach-msg-rcv-resource-shortage})[\alpha]
\end{array}$$

2.7 Message Descriptors

Our formalization of a Mach kernel state in [BS94b] describes the contents of a message as a indexed list, elements of which are either transit data, a transit port right, or a transit null item.

The user interface to kernel IPC services includes a language for a sending task to give the kernel instructions on both the intended contents of the message to be sent and the side effects the sender intends to observe as a result. For the receiver, the same language describes the effects on the receiver of receiving the message. The instructions are encoded in the task's message buffer. We call the instructions encoded in the message buffer a *message descriptor*. The message descriptor is a specification artifact that allows us to ignore the details of the layout of the encoding in the task's address space.

To summarize, have three distinct concepts:

1. The message buffer, which is a portion of the calling task's address space. This is outside of our specification.

2. The message descriptor, which are the instructions encoded in the message buffer.

3. The message, which is constructed in the kernel state according to the message descriptor, using the resources of the calling task.

Syntax of message descriptors

We define a specification predicate Mdp which describes the syntax of an abstract message descriptor. Each element of a message descriptor is one of:

- A tuple $\langle$`'right`, n, $instr$, $r\rangle$ instructing how a transit right should be created.

- A tuple $\langle$`'data`, a, va, l, $instr\rangle$, instructing the kernel to create a transit memory.

- A tuple $\langle$`'null`, $instr\rangle$, which instructs the kernel to include a null message element.

$$\begin{aligned}&\mathrm{Mdp}\,(md)\\ \equiv\ &\forall\, 0 \le i < |md|\colon\ (\text{Md-Rightp}\,(md_i) \lor \text{Md-Datap}\,(md_i) \lor \text{Md-Nullp}\,(md_i))\end{aligned}$$

$$\begin{aligned}
&\text{Md-Rightp}\,(md_i)\\
\equiv\ &\exists\, n \in \mathcal{N},\ instr \in \{\texttt{'move},\ \texttt{'copy},\ \texttt{'make}\},\ r \in \mathcal{R}\text{:}\\
&\quad (md_i = \langle \texttt{'right},\ n,\ instr,\ r\rangle)[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Md-Datap}\,(md_i)\\
\equiv\ &\exists\, a \in \{\texttt{'in-line},\ \texttt{'out-of-line}\},\ \texttt{0} \le va < \textsc{address-space-limit},\\
&\qquad \texttt{0} \le l < \textsc{address-space-limit},\ instr \in \{\texttt{'delete},\ \texttt{'no-delete}\}\text{:}\\
&\quad (md_i = \langle \texttt{'data},\ a,\ va,\ l,\ instr\rangle)[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Md-Nullp}\,(md_i)\\
\equiv\ &\exists\, instr \in \{\texttt{'null-right},\ \texttt{'null-memory},\ \texttt{'dead-right}\}\text{:}\\
&\quad (md_i = \langle \texttt{'null},\ instr\rangle)[\alpha]
\end{aligned}$$

2.8 Extracting resources from a task

A part of sending a message is creating the message from the resources of a task, according to the instructions in a message descriptor. The same mechanism is used to process other arguments.

The temporal predicate Md-To-Messagep recognizes a computation in which a message is created from a task. The low-level operations on port rights and address spaces are described in Chapter 7.

$$
\begin{array}{ll}
 & \text{Md-To-Messagep}\,(md,\ l,\ mg) \\
\equiv & \quad \forall\, \mathtt{0} \le i < l\text{: Md-El-To-Messagep}\,(md,\ mg,\ i) \\
 & \wedge\ \Diamond(\text{message-size}\,(mg) = l)[\alpha]
\end{array}
$$

The predicate Md-El-To-Messagep summarizes the posible activities in constructing a message element from a message descriptor element. The possibilities are grouped into three classes by the type of message element constructed.

Null message elements: null-right, dead-right, or null-memory.

Transit rights: make-send, copy-send, move-send, move-send-once, make-send-once, or move-receive.

Transit memories: in line, no delete; out of line, delete; and out of line, no delete.

$$
\begin{array}{ll}
 & \text{Md-El-To-Messagep}\,(md,\ mg,\ i) \\
\equiv & \quad \text{Md-Nulls-To-Messagep}\,(md,\ mg,\ i) \\
 & \vee\ \text{Md-Rights-To-Messagep}\,(md,\ mg,\ i) \\
 & \vee\ \text{Md-Memories-To-Messagep}\,(md,\ mg,\ i)
\end{array}
$$

Null message elements

A null message descriptor element can be tagged `'null-right`, `'dead-right`, or `'null-memory`. It causes the creation of a null message element with the same tag. A null message element can also be created when the message descriptor specifies certain port right operations where the local name is a dead right or when a zero-length out-of-line memory region is specified.

$$
\begin{array}{l}
\quad \text{Md-Nulls-To-Messagep}\,(md,\ mg,\ i) \\
\equiv \exists\ tag \in \{\texttt{'null-right},\ \texttt{'null-memory},\ \texttt{'dead-right}\}\text{:} \\
\quad (\quad (md_i = \langle \texttt{'null},\ tag\rangle)[\alpha] \\
\quad\ \ \wedge\ \Diamond\uparrow\text{null-message-element-rel}\,(mg,\ tag,\ i)[\alpha])
\end{array}
$$

Port rights

A port right is extracted from the sending task's port name space from a given local name n, right type $r \in (\ \mathcal{R} = \{\texttt{'receive},\ \texttt{'send},\ \texttt{'send-once}\})$, and instruction *instr* $\in \{\texttt{'copy},\ \texttt{'move},\ \texttt{'make}\}$. Transit rights and the send port and reply port (if any) are all processed by this mechanism. The `'move` instruction causes a side effect in the sending task — the right is removed. The other instructions cause a new right to be cloned from an existing right[3].

Creating a message element

The predicate Md-Rights-To-Messagep recognizes the transfer of a port right from a task into a message element. A null message element can be constructed from a dead right.

$$
\begin{array}{l}
\quad \text{Md-Rights-To-Messagep}\,(md,\ mg,\ i) \\
\equiv \exists\ n \in \mathcal{N},\ instr \in \{\texttt{'make},\ \texttt{'copy},\ \texttt{'move}\},\ r \in \mathcal{R}\text{:} \\
\quad (\quad (md_i = \langle \texttt{'right},\ n,\ instr,\ r\rangle)[\alpha] \\
\quad\ \ \wedge\ (\quad \exists\ p \in \text{ALL-ENTITIES:} \\
\qquad\qquad\quad (\quad \text{Extract-Port-Right}\,(\mathbf{t},\ n,\ instr,\ r,\ p) \\
\qquad\qquad\quad\ \ ;\ \uparrow\text{transit-right-rel}\,(mg,\ p,\ r,\ i)[\alpha]) \\
\qquad\quad\ \ \vee\ (\quad \text{Extract-Dead-Right}\,(\mathbf{t},\ n,\ instr,\ r) \\
\qquad\qquad\ \ ;\ \uparrow\text{null-message-element-rel}\,(mg,\ \texttt{'dead-right},\ i)[\alpha])))
\end{array}
$$

Send destination and reply ports

The destination and reply ports for a sent message are processed with the same mechanism as for transit send rights. In the implementation,

[3] The processing of a message element should "see" the side effects of previous message elements. Examples: The second move-receive for a given port should fail; two move-send operations should result in the refcount being decremented twice. These are safety properties that are not stated here.

these ports are specified by fields in the message buffer header; in our formalization, they are specified by separate arguments.

The function Extract-Send-Right is like Extract-Port-Right except that the `'receive` case is not allowed. The user can also specify that no reply port is desired.

$$\text{Extract-Destination}\,(n,\ \mathit{instr},\ r,\ p) \equiv \text{Extract-Send-Right}\,(n,\ \mathit{instr},\ r,\ p)$$

$$\begin{array}{lll}
 & \multicolumn{2}{l}{\text{Extract-Reply}\,(n,\ \mathit{instr},\ r,\ p)} \\
\equiv & & \text{Extract-Send-Right}\,(n,\ \mathit{instr},\ r,\ p) \\
 & \lor & \quad (n = \textsc{nullname})[\alpha] \\
 & & \land\ (\langle \mathit{instr},\ r\rangle = \langle \texttt{'none},\ \texttt{'none}\rangle)[\alpha] \\
 & & \land\ \Diamond\!\uparrow\!(p = \textsc{null-ptr})[\alpha]
\end{array}$$

Receive port

The argument *rcv-name* to *mach-msg-receivep* identifies the port or port set from which the message will be received. If it is the name of a port set, the kernel can choose any port in the set[4]. If the name does not name a port set, it must be a receive right which is not in a port set.

$$\begin{array}{lll}
 & \multicolumn{2}{l}{\text{Names-Receiving-Port}\,(n,\ p)} \\
\equiv & & \Diamond\exists\, n_1 \in \mathcal{N}\text{:} \\
 & & \quad (\quad \text{in-port-set}\,(\mathbf{t},\ n_1)[\alpha] \\
 & & \quad\ \ \land\ (n = \text{holding-port-set-name}\,(\mathbf{t},\ n_1))[\alpha] \\
 & & \quad\ \ \land\ (\text{named-port}\,(\mathbf{t},\ n_1) = p)[\alpha]) \\
 & \lor & \Diamond(\quad \text{r-right}\,(\mathbf{t},\ n)[\alpha] \\
 & & \quad \land\ (\text{named-port}\,(\mathbf{t},\ n) = p)[\alpha] \\
 & & \quad \land\ \neg\ \text{in-port-set}\,(\mathbf{t},\ n)[\alpha])
\end{array}$$

Memories

Transit memories are extracted from the sending task's address space. The data may be flagged either `'in-line` or `'out-of-line`. The

[4]The implementation insures that messages received from a port set are received in the order they were enqueued. This specification only requires that the messages from each port be received in order.

`'delete` instruction causes the data to be deleted from the address space as a side effect.

The implementation of transit memories is extremely complex. The Mach implementation is well-known for its ability to transfer large blocks of memory efficiently by means of copy-on-write optimizations[5]. For the purposes of this specification, these implementation issues are not relevant[6].

$$
\begin{array}{ll}
 & \text{Md-Memories-To-Messagep}\,(md,\ mg,\ i) \\
\equiv & \quad\ \ \text{Md-In-Line-No-Delete-To-Messagep}\,(md,\ mg,\ i) \\
 & \lor\ \text{Md-Out-Of-Line-No-Delete-To-Messagep}\,(md,\ mg,\ i) \\
 & \lor\ \text{Md-Out-Of-Line-Delete-To-Messagep}\,(md,\ mg,\ i)
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Md-In-Line-No-Delete-To-Messagep}\,(md,\ mg,\ i) \\
\equiv & \exists\ \mathtt{0} \le va < \textsc{address-space-limit},\ \mathtt{1} \le l < (\textsc{address-space-limit} - va): \\
 & \quad (\quad (md_i = \langle \mathtt{'data},\ \mathtt{'in\text{-}line},\ va,\ l,\ \mathtt{'no\text{-}delete}\rangle)[\alpha] \\
 & \quad\ \ \land\ \text{Va-Region-To-Messagep}\,(va,\ mg,\ \mathtt{'in\text{-}line},\ \mathtt{0},\ l,\ i))
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Md-Out-Of-Line-No-Delete-To-Messagep}\,(md,\ mg,\ i) \\
\equiv & \exists\ \mathtt{0} \le va < \textsc{address-space-limit},\ \mathtt{1} \le l < (\textsc{address-space-limit} - va): \\
 & \quad (\quad (md_i = \langle \mathtt{'data},\ \mathtt{'out\text{-}of\text{-}line},\ va,\ l,\ \mathtt{'no\text{-}delete}\rangle)[\alpha] \\
 & \quad\ \ \land\ \text{Va-Region-To-Messagep}\,(va,\ mg,\ \mathtt{'out\text{-}of\text{-}line}, \\
 & \qquad\qquad\qquad\qquad\qquad va - \text{trunc-page}\,(va),\ l,\ i))
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Md-Out-Of-Line-Delete-To-Messagep}\,(md,\ mg,\ i) \\
\equiv & \exists\ \mathtt{0} \le va < \textsc{address-space-limit},\ \mathtt{1} \le l < (\textsc{address-space-limit} - va): \\
 & \quad (\quad (md_i = \langle \mathtt{'data},\ \mathtt{'out\text{-}of\text{-}line},\ va,\ l,\ \mathtt{'delete}\rangle)[\alpha] \\
 & \quad\ \ \land\ \text{Va-Region-To-Messagep}\,(va,\ mg,\ \mathtt{'out\text{-}of\text{-}line}, \\
 & \qquad\qquad\qquad\qquad\qquad va - \text{trunc-page}\,(va),\ l,\ i) \\
 & \quad\ \ \land\ \text{All-Eventually-Not-Allocated}\,(\mathtt{t},\ va,\ l))
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Va-Region-To-Messagep}\,(va,\ mg,\ tag,\ o,\ l,\ i) \\
\equiv & \exists\ m \in \textsc{all-entities}: \\
 & \quad (\Diamond(\quad \uparrow\text{memoryp}\,(m)[\alpha] \\
 & \qquad\ \land\ \Diamond\uparrow\text{temporary-rel}\,(m)[\alpha] \\
 & \qquad\ \land\ \text{Extract-Va-Region}\,(\mathtt{t},\ va,\ o,\ l,\ m) \\
 & \qquad\ \land\ \Diamond\uparrow\text{transit-memory-rel}\,(mg,\ m,\ tag,\ o,\ l,\ i)[\alpha]))
\end{array}
$$

[5]See [BS94b] for a discussion of our model of transit memories.

[6]In fact, the details of the optimization are visible to the user. There are security implications, for example.

2.9 Inserting resources into a task

Receiving a message causes rights and data to become a part of the receiving task's resources. The same mechanism is used on a send operation when it times out or is interrupted. This "pseudo-receive" operation readies the message buffer and its contents for a retry. The predicate Message-To-Mdp recognizes a computation in which a message's contents are given to a task, and a message descriptor is constructed.

$$\text{Message-To-Mdp}\,(mg,\ md) \equiv \forall\ \mathtt{0} \leq i < |md|\text{:}\ \ \text{Message-To-Md-Elp}\,(mg,\ md,\ i)$$

$$\begin{array}{ll} & \text{Message-To-Md-Elp}\,(mg,\ md,\ i) \\ \equiv & \quad \text{Message-Null-To-Md-Elp}\,(mg,\ md,\ i) \\ & \lor\ \text{Message-Right-To-Md-Elp}\,(mg,\ md,\ i) \\ & \lor\ \text{Message-Memory-To-Md-Elp}\,(mg,\ md,\ i) \end{array}$$

Null message elements

Receiving a null message element causes no side effects in the receiving task. Only the message descriptor is affected.

$$\begin{array}{lll} & \text{Message-Null-To-Md-Elp}\,(mg,\ md,\ i) & \\ \equiv & \qquad \text{null-message-element-rel}\,(mg,\ \mathtt{'null\text{-}right},\ i)[\alpha] & \\ & \quad \land\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'null},\ \mathtt{'null\text{-}right} \rangle)[\alpha] & \\ & \lor \qquad \text{null-message-element-rel}\,(mg,\ \mathtt{'null\text{-}memory},\ i)[\alpha] & \\ & \quad \land\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'null},\ \mathtt{'null\text{-}memory} \rangle)[\alpha] & \\ & \lor \qquad \text{null-message-element-rel}\,(mg,\ \mathtt{'dead\text{-}right},\ i)[\alpha] & \\ & \quad \land\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'null},\ \mathtt{'dead\text{-}right} \rangle)[\alpha] & \end{array}$$

Port rights

When copying a message into a task, the new resources are inserted and a message descriptor is constructed to describe the activity. A received message descriptor is simpler than a sent one in that there are fewer choices: all rights are marked with the instruction `'move`.

$$
\begin{array}{lll}
& \text{Message-Right-To-Md-Elp}\,(mg,\ md,\ i) & \\
\equiv & \exists\ p \in \textsc{all-entities},\ n \in \mathcal{N},\ r \in \mathcal{R}\text{:} & \\
& (\quad \text{transit-right-rel}\,(mg,\ p,\ r,\ i)[\alpha] & \\
& \ \ \wedge\ \text{Insert-Port-Right}\,(\mathtt{t},\ p,\ n,\ r) & \\
& \ \ \wedge\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'right},\ n,\ \mathtt{'move},\ r\rangle)[\alpha]) &
\end{array}
$$

The reply port (if any) is inserted into the receiver's name space in the same way as transit rights.

$$
\begin{array}{ll}
& \text{Insert-Reply-Port}\,(n,\ r,\ p) \\
\equiv & \Diamond(\quad (p = \textsc{null-ptr})[\alpha] \wedge (n = \textsc{nullname})[\alpha] \\
& \ \ \vee\ \text{Insert-Send-Right}\,(\mathtt{t},\ p,\ n) \wedge (r = \mathtt{'send})[\alpha] \\
& \ \ \vee\ \text{Insert-Send-Once-Right}\,(\mathtt{t},\ p,\ n) \wedge (r = \mathtt{'send\text{-}once})[\alpha])
\end{array}
$$

Memories

The predicate Message-Memory-To-Md-Elp describes how a transit memory is inserted into the receiving task's address space.

In line data is copied into the receiver's message buffer[7].

Out of line data is mapped into an unallocated region of the receiver's address space. In the message descriptor, all out-of-line memory segments are marked 'delete.

$$
\begin{array}{ll}
& \text{Message-Memory-To-Md-Elp}\,(mg,\ md,\ i) \\
\equiv & \exists\ m \in \textsc{all-entities},\ 0 \le o < \textsc{pagesize},\ 0 \le va < \textsc{address-space-limit}, \\
& \quad 1 \le l < (\textsc{address-space-limit} - va)\text{:} \\
& (\qquad \text{transit-memory-rel}\,(mg,\ m,\ \mathtt{'in\text{-}line},\ o,\ l,\ i)[\alpha] \\
& \qquad \wedge\ \text{Insert-In-Line-Data}\,(\mathtt{t},\ va,\ m,\ o,\ l) \\
& \qquad \wedge\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'data},\ \mathtt{'in\text{-}line},\ va,\ l,\ \mathtt{'no\text{-}delete}\rangle)[\alpha] \\
& \quad \vee\quad \text{transit-memory-rel}\,(mg,\ m,\ \mathtt{'out\text{-}of\text{-}line},\ o,\ l,\ i)[\alpha] \\
& \qquad \wedge\ \text{Insert-Out-Of-Line-Data}\,(\mathtt{t},\ va,\ m,\ o,\ l) \\
& \qquad \wedge\ \Diamond\!\uparrow\!(md_i = \langle \mathtt{'data},\ \mathtt{'out\text{-}of\text{-}line},\ va,\ l,\ \mathtt{'delete}\rangle)[\alpha])
\end{array}
$$

[7]We don't require that the given virtual address be within the message buffer.

Chapter 3

Port Interface

3.1 mach_port_allocate

DESCRIPTION

Create a new right in the target task: either a receive right, an empty port set, or a dead name.

PARAMETERS

t. The target task.

right. A flag indicating which right is requested, one of {`'receive`, `'dead-right`, `'port-set`}.

n. [out] The returned local port name.

OUTCOMES

There are five possible outcomes: *success*, *invalid task*, *invalid value*, *no space*, and *resource shortage*.

$$
\begin{array}{ll}
 & \text{Mach-Port-Allocatep} \\
\equiv & \quad \text{Mach-Port-Allocate-Success} \\
 & \lor\ \text{Mach-Port-Allocate-Invalid-Task} \\
 & \lor\ \text{Mach-Port-Allocate-Invalid-Value} \\
 & \lor\ \text{Mach-Port-Allocate-No-Space} \\
 & \lor\ \text{Mach-Port-Allocate-Resource-Shortage}
\end{array}
$$

SPECIFICATION

In a successful outcome, either a new receive right is created, an empty port set is created, or a dead name is created. Initially, the target task must exist, and **n** is not a local name for the target task.

Mach-Port-Allocate-Success

$$\begin{array}{ll} \equiv & \quad \Diamond \text{taskp}\,(\mathbf{t})[\alpha] \\ & ; \ \Diamond(\neg \ \text{local-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha] \\ & ; \quad \text{Mach-Port-Allocate-Receive} \\ & \quad \vee \ \text{Mach-Port-Allocate-Port-Set} \\ & \quad \vee \ \text{Mach-Port-Allocate-Dead-Right} \\ & ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

When **right** = `'receive`, a port is created and a receive right is established.

Mach-Port-Allocate-Receive

$$\begin{array}{l} \equiv \exists\, p \in \textsc{all-entities}: \\ \quad (\quad \Diamond(\mathbf{right} = \texttt{'receive})[\alpha] \\ \quad\ \ ; \ \Diamond\uparrow\text{portp}\,(p)[\alpha] \\ \quad\ \ ; \ \Diamond\text{New-Receive-Right}\,(\mathbf{t},\ p,\ \mathbf{n})) \end{array}$$

When **right** =`'port-set`, an empty port set is established for the target task.

Mach-Port-Allocate-Port-Set

$$\equiv \Diamond(\mathbf{right} = \texttt{'port-set})[\alpha] \ ; \ \Diamond\text{New-Port-Set}\,(\mathbf{t},\ \mathbf{n})$$

When **right** =`'dead-right`, name **n** becomes a dead name for the target task. The new dead right has reference count one.

Mach-Port-Allocate-Dead-Right

$$\equiv \Diamond(\mathbf{right} = \texttt{'dead-right})[\alpha] \ ; \ \Diamond\text{New-Dead-Right}\,(\mathbf{t},\ \mathbf{n},\ \texttt{1})$$

An *invalid task* outcome results when the target task argument is discovered not to be a task. The *invalid value* outcome results when the **right** argument has a bad value.

Mach-Port-Allocate-Invalid-Task

$$\equiv \Diamond(\neg \ \text{taskp}\,(\mathbf{t}))[\alpha] \ ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]$$

Mach-Port-Allocate-Invalid-Value

$$\begin{array}{ll} \equiv & \quad \Diamond(\mathbf{right} \notin \{\texttt{'receive},\ \texttt{'port-set},\ \texttt{'dead-name}\})[\alpha] \\ & ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha] \end{array}$$

We specify nothing about the computation in the case of the *no space* and *resource-shortage* outcomes, other than the setting of the return code.

Mach-Port-Allocate-No-Space $\equiv \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-no-space})[\alpha]$

Mach-Port-Allocate-Resource-Shortage
$\equiv \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha]$

3.2 mach_port_allocate_name

DESCRIPTION

Create a new right in the target task: either a receive right, an empty port set, or a dead name. The new local name is chosen by the user, not picked by the kernel.

PARAMETERS

t. The target task.

right. A flag indicating which right is requested.

n. The name the caller wishes to be used for the new right.

OUTCOMES

$$\begin{aligned}
&\text{Mach-Port-Allocate-Namep}\\
\equiv\quad &\text{Mach-Port-Allocate-Success}\\
\lor\ &\text{Mach-Port-Allocate-Invalid-Task}\\
\lor\ &\text{Mach-Port-Allocate-Invalid-Value}\\
\lor\ &\text{Mach-Port-Allocate-Name-Exists}\\
\lor\ &\text{Mach-Port-Allocate-No-Space}\\
\lor\ &\text{Mach-Port-Allocate-Resource-Shortage}
\end{aligned}$$

SPECIFICATION

The specification for `mach_port_allocate_name` is almost identical to the one for `mach_port_allocate`. The only difference is that the name parameter in `mach_port_allocate_name` is an input, and so there exists an additional return code indicating that the name is already in use.

An *name exists* outcome results when **name** is already in use within the target task.

$$\begin{aligned}
&\text{Mach-Port-Allocate-Name-Exists}\\
\equiv\ &\Diamond\text{local-namep}\,(\mathbf{t}, \mathbf{n})[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-name-exists})[\alpha]
\end{aligned}$$

3.3 mach_port_deallocate

DESCRIPTION

Decrement the reference count on a send right, send-once right, or dead name. If the reference count becomes zero, remove the right.

PARAMETERS

t. The target task.

n. The name of the right.

OUTCOMES

There are four possible outcomes: *success*, *invalid task*, *invalid name*, and *invalid right*.

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{Mach-Port-Deallocatep}} \\
\equiv & & \text{Mach-Port-Deallocate-Success} \\
 & \lor & \text{Mach-Port-Deallocate-Invalid-Task} \\
 & \lor & \text{Mach-Port-Deallocate-Invalid-Name} \\
 & \lor & \text{Mach-Port-Deallocate-Invalid-Right}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the reference count for a send right, send-once right, or dead right is decremented by 1. In a legal kernel state, a name can represent at most one of a send, send-once or dead right. Section 7.2.2 contains detailed specifications for decrementing rights.

$$
\begin{array}{llll}
 & \multicolumn{3}{l}{\text{Mach-Port-Deallocate-Success}} \\
\equiv & \multicolumn{3}{l}{\Diamond\text{taskp}\,(\mathbf{t})[\alpha]} \\
 & ; & & \text{Deallocate-Send-Right}\,(\mathbf{t},\, \mathbf{n},\, \mathtt{1}) \\
 & & \lor & \text{Remove-Send-Once-Right}\,(\mathbf{t},\, \mathbf{n}) \\
 & & \lor & \text{Deallocate-Dead-Right}\,(\mathbf{t},\, \mathbf{n},\, \mathtt{1}) \\
 & ; & \multicolumn{2}{l}{\Diamond\uparrow(\mathbf{rc} = \mathtt{'kern\text{-}success})[\alpha]}
\end{array}
$$

An *invalid task* outcome results when the target task argument is discovered not to be a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The *invalid right* outcome results when name **n** is a local name in the target task, but is not a send right, send-once right, or dead right.

$$\begin{array}{ll} & \text{Mach-Port-Deallocate-Invalid-Task} \\ \equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha] \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Mach-Port-Deallocate-Invalid-Name}} \\ \equiv & & \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \\ & ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha] \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Mach-Port-Deallocate-Invalid-Right}} \\ \equiv & & \Diamond(\neg\ (\text{s-right}\,(\mathbf{t},\, \mathbf{n}) \vee \text{so-right}\,(\mathbf{t},\, \mathbf{n}) \vee \text{dead-right-namep}\,(\mathbf{t},\, \mathbf{n})))[\alpha] \\ & ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha] \end{array}$$

3.4 mach_port_destroy

DESCRIPTION

Remove a local name from a task's port name space.

PARAMETERS

t. The target task.

n. The local name to be removed.

OUTCOMES

There are three possible outcomes: *success*, *invalid task*, and *invalid name*.

$$\begin{array}{ll} & \text{Mach-Port-Destroyp} \\ \equiv & \quad \text{Mach-Port-Destroy-Success} \\ & \vee\ \text{Mach-Port-Destroy-Invalid-Task} \\ & \vee\ \text{Mach-Port-Destroy-Invalid-Name} \end{array}$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the local name is destroyed[1]. If the name denotes a receive right, the port associated with the name is terminated. For a discussion of the recursive nature of entity destruction in Mach, see Section 7.4.1.

$$\begin{array}{ll} & \text{Mach-Port-Destroy-Success} \\ \equiv & \quad \text{taskp}\,(\mathbf{t})[\alpha] \\ & ;\quad \quad \text{Mach-Port-Destroy-Port-Right} \\ & \quad\ \ \vee\ \text{Mach-Port-Destroy-Port-Set} \\ & \quad\ \ \vee\ \text{Mach-Port-Destroy-Dead-Right} \\ & ;\ \ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

[1]If the right is a send right but not a receive right, we may send a no-more-senders notification. If it is a send-once right, we send a send-once notification. We do not model this activity.

Mach-Port-Destroy-Port-Right

$$\begin{array}{ll} \equiv & \Diamond\text{port-right-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha] \\ ; & \Diamond(\text{r-right}\,(\mathbf{t},\,\mathbf{n})[\alpha] \rightarrow \text{Terminate-Port}\,(\text{named-port}\,(\mathbf{t},\,\mathbf{n}))) \\ ; & \Diamond{\downarrow}\text{local-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha] \end{array}$$

Mach-Port-Destroy-Port-Set

$$\equiv \Diamond\text{port-set-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha]$$

Mach-Port-Destroy-Dead-Right

$$\equiv \Diamond\text{dead-right-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha]$$

An *invalid task* outcome results when the target task argument is discovered not to be a task. The *invalid name* outcome results when name **n** is not a local name in the target task.

Mach-Port-Destroy-Invalid-Task

$$\equiv \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]$$

Mach-Port-Destroy-Invalid-Name

$$\begin{array}{ll} \equiv & \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t},\,\mathbf{n}))[\alpha] \\ ; & \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha] \end{array}$$

3.5 mach_port_extract_right

DESCRIPTION

Transfer a port right. The effect is equivalent to forcing the target task to send a port right to the calling task.

PARAMETERS

- **ct**. The invoking (current) task. This argument is implicit in the implementation.
- **t**. The target task.
- $\mathbf{n_1}$. The name of interest in **t**'s local name space.
- **instr**. The method for extracting the right from the target task, one of {`'make`, `'copy`, `'move`}.
- **r**. The type of right to be extracted, one of {`'send`, `'receive`, `'send-once`}. In the implementation, **instr** and **r** are encoded in a single argument *dtype*.
- $\mathbf{n_2}$. [out] The new local name in **ct**'s local name space.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, *invalid value*, and *invalid right*.

$$\begin{aligned} & \text{Mach-Port-Extract-Rightp} \\ \equiv \quad & \quad \text{Mach-Port-Extract-Right-Success} \\ & \lor\ \text{Mach-Port-Extract-Right-Invalid-Task} \\ & \lor\ \text{Mach-Port-Extract-Right-Invalid-Name} \\ & \lor\ \text{Mach-Port-Extract-Right-Invalid-Value} \\ & \lor\ \text{Mach-Port-Extract-Right-Invalid-Right} \end{aligned}$$

SPECIFICATION

A successful outcome follows the semantics of a send of this right followed by a receive.

$$
\begin{array}{ll}
& \text{Mach-Port-Extract-Right-Success} \\
\equiv & \quad \exists\, p \in \textsc{all-entities}: \\
& \quad\quad (\ \ \Diamond\text{Extract-Port-Right}\,(\mathbf{t},\ \mathbf{n_1},\ \mathbf{instr},\ \mathbf{r},\ p) \\
& \quad\quad\quad ;\ \text{Insert-Port-Right}\,(\mathbf{ct},\ p,\ \mathbf{n_2},\ \mathbf{r})) \\
& \quad \vee\ (\ \ \Diamond\text{Extract-Dead-Right}\,(\mathbf{t},\ \mathbf{n_1},\ \mathbf{instr},\ \mathbf{r}) \\
& \quad\quad ;\ \uparrow(\mathbf{n_2} = \textsc{deadname})[\alpha]) \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The outcome *invalid value* results when the **instr** and **r** arguments are not a legal combination, and the *invalid right* outcome results when the arguments **name**, **instr**, and **r** are not a legal combination.

$$
\begin{array}{ll}
& \text{Mach-Port-Extract-Right-Invalid-Task} \\
\equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Extract-Right-Invalid-Name} \\
\equiv & \quad \Diamond(\text{taskp}\,(\mathbf{t})[\alpha] \wedge (\neg\ \text{local-namep}\,(\mathbf{t},\ \mathbf{n_1}))[\alpha]) \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Extract-Right-Invalid-Value} \\
\equiv & \quad \Diamond\neg\ \text{Legal-Instr-Right}\,(\mathbf{instr},\ \mathbf{r}) \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Extract-Right-Invalid-Right} \\
\equiv & \quad \Diamond\text{Legal-Instr-Right}\,(\mathbf{instr},\ \mathbf{r}) \\
& ;\ \Diamond\neg\ \text{Legal-Name-Instr-Right-Deadok}\,(\mathbf{t},\ \mathbf{n_1},\ \mathbf{instr},\ \mathbf{r}) \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{array}
$$

3.6 mach_port_get_refs

DESCRIPTION

Look up the reference count for a local name. If the name is a local name but not of the specified type, the returned count is zero.

PARAMETERS

- **t**. The target task.
- **n**. A local name in **t**.
- **type**. The type of right of interest, one of `'send`, `'receive`, `'send-once`, `'port-set`, or `'dead-name`.
- **i**. [out] The returned count.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, and *invalid right*.

$$\begin{array}{ll} & \text{Mach-Port-Get-Refsp} \\ \equiv & \quad \text{Mach-Port-Get-Refs-Success} \\ & \vee\ \text{Mach-Port-Get-Refs-Invalid-Task} \\ & \vee\ \text{Mach-Port-Get-Refs-Invalid-Name} \\ & \vee\ \text{Mach-Port-Get-Refs-Invalid-Right} \end{array}$$

SPECIFICATION

The success outcome occurs when **n** is a local name in **t**, and **type** is one of the expected types. If the name **n** is not of the expected type, the returned **i** is zero. For `'send-once`, `'receive`, or `'port-set`, the returned **i** is always zero or one. A send right coalesces with a receive right for the same port. The send right reference is adjusted if there is also a receive right.

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Refs-Success} \\
\equiv & (\quad (\mathbf{type} = \texttt{'send})[\alpha] \wedge \text{Mach-Port-Get-Send-Refs} \\
& \quad \vee\ (\mathbf{type} = \texttt{'receive})[\alpha] \wedge \text{Mach-Port-Get-Receive-Refs} \\
& \quad \vee\ \quad (\mathbf{type} = \texttt{'send-once})[\alpha] \\
& \qquad \wedge\ \text{Mach-Port-Get-Send-Once-Refs} \\
& \quad \vee\ (\mathbf{type} = \texttt{'port-set})[\alpha] \wedge \text{Mach-Port-Get-Port-Set-Refs} \\
& \quad \vee\ \quad (\mathbf{type} = \texttt{'dead-name})[\alpha] \\
& \qquad \wedge\ \text{Mach-Port-Get-Dead-Name-Refs}) \\
& \wedge\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Send-Refs} \\
\equiv & \quad \Diamond(\quad \text{s-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \\
& \qquad \wedge\ (\neg\ \text{r-right}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \\
& \qquad \wedge\ \Diamond\!\uparrow\!(\mathbf{i} = \text{port-right-refcount}\,(\mathbf{t},\, \mathbf{n}))[\alpha]) \\
& \vee\ \Diamond(\quad \text{s-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \\
& \qquad \wedge\ \text{r-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \\
& \qquad \wedge\ \Diamond\!\uparrow\!(\mathbf{i} = \text{port-right-refcount}\,(\mathbf{t},\, \mathbf{n}) - \texttt{1})[\alpha]) \\
& \vee\ \Diamond((\neg\ \text{s-right}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{0})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Receive-Refs} \\
\equiv & \Diamond(\quad \text{r-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{1})[\alpha] \\
& \quad \vee\ (\neg\ \text{r-right}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{0})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Send-Once-Refs} \\
\equiv & \Diamond(\quad \text{so-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{1})[\alpha] \\
& \quad \vee\ (\neg\ \text{so-right}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{0})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Port-Set-Refs} \\
\equiv & \Diamond(\quad \text{port-set-namep}\,(\mathbf{t},\, \mathbf{n})[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{1})[\alpha] \\
& \quad \vee\ (\neg\ \text{port-set-namep}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{0})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Get-Dead-Name-Refs} \\
\equiv & \Diamond(\quad \text{dead-right-rel}\,(\mathbf{t},\, \mathbf{n},\, \mathbf{i})[\alpha] \\
& \quad \vee\ (\neg\ \text{dead-right-namep}\,(\mathbf{t},\, \mathbf{n}))[\alpha] \wedge \Diamond\!\uparrow\!(\mathbf{i} = \texttt{0})[\alpha])
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The *invalid right* outcome results when the **type** argument is not a legal value.

$$\begin{aligned}
&\text{Mach-Port-Get-Refs-Invalid-Task}\\
\equiv\ &\Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Get-Refs-Invalid-Name}\\
\equiv\ &\quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t}, \mathbf{n}))[\alpha]\\
&\ ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Get-Refs-Invalid-Right}\\
\equiv\ &\quad \Diamond(\mathbf{type} \notin \{\texttt{'send}, \texttt{'receive}, \texttt{'send-once}, \texttt{'port-set},\\
&\qquad\qquad\qquad \texttt{'dead-name}\})[\alpha]\\
&\ ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{aligned}$$

3.7 mach_port_get_set_status

DESCRIPTION

Return the members of a port set, found at some time during the computation. This call returns the target task's names for the receive rights, not the calling task's. There is no guarantee that this set is accurate upon return to the calling task.

PARAMETERS

- **t**. The target task.
- **n**. The local name of a port set.
- **N**. [out] The returned list of names[2].

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, *invalid right*, and *resource shortage.*

$$
\begin{array}{ll}
 & \text{Mach-Port-Get-Set-Statusp} \\
\equiv & \quad \text{Mach-Port-Get-Set-Status-Success} \\
 & \lor\ \text{Mach-Port-Get-Set-Status-Invalid-Task} \\
 & \lor\ \text{Mach-Port-Get-Set-Status-Invalid-Name} \\
 & \lor\ \text{Mach-Port-Get-Set-Status-Invalid-Right} \\
 & \lor\ \text{Mach-Port-Get-Set-Status-Resource-Shortage}
\end{array}
$$

SPECIFICATION

The success outcome occurs when **n** is the name of a port set in **t**. The returned list **N** is the list of names in the port set.

$$
\begin{array}{ll}
 & \text{Mach-Port-Get-Set-Status-Success} \\
\equiv & \Diamond\text{port-set-rel}\,(\mathbf{t},\,\mathbf{n},\,\mathbf{N})[\alpha]\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

[2]In the implementation, the list **N** is returned in an out-of-line memory block.

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The *invalid right* outcome results when the **type** argument is a local name but is not a port set. We say nothing about a resource shortage.

$$
\begin{array}{ll}
 & \text{Mach-Port-Get-Set-Status-Invalid-Task} \\
\equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{array}
$$

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{Mach-Port-Get-Set-Status-Invalid-Name}} \\
\equiv & & \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha] \\
 & ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{array}
$$

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{Mach-Port-Get-Set-Status-Invalid-Right}} \\
\equiv & & \Diamond(\text{local-namep}\,(\mathbf{t},\ \mathbf{n})[\alpha] \wedge (\neg\ \text{port-set-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha]) \\
 & ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Get-Set-Status-Resource-Shortage} \\
\equiv & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha]
\end{array}
$$

3.8 mach_port_insert_right

DESCRIPTION

Extract a port right from the calling task and insert it into the target task. The result is *almost* equivalent to the target task receiving a message containing the transit right from the calling task – the difference is that the calling task can choose the new name for the target task. Flag arguments tells the type of right to be constructed and the side effect on the calling task.

PARAMETERS

- **t**. The target task.
- $\mathbf{n_2}$. The new local name to be inserted into **t**.
- **ct**. The invoking (current) task. In the implementation, this argument is implicit.
- $\mathbf{n_1}$. A local name in **ct**'s name space.
- **instr**. An instruction for disposition of $\mathbf{n_1}$, one of `'move`, `'copy`, or `'make`.
- **r**. The type of right desired for $\mathbf{n_2}$, either `'send`, `'receive`, or `'send-once`. In the implementation, **instr** and **r** are encoded in a single argument *right_type*.

OUTCOMES

The possible outcomes are: *success*, *invalid task*, *invalid value*, *name exists*, *invalid capability*, *urefs overflow*, *right exists*, *resource shortage*[3]. We say nothing about the latter.

The kernel service `mach_port_insert_right` operates in two distinct phases. First, the right is extracted from the calling task. (This phase

[3]The Mach 3.0 code and the OSF documentation [Loe91] seem to disagree on the exact meanings of the error return codes. We model the code.

is implemented by the MIG interface.) If this phase is successful, the right is inserted into the target task. If the second phase fails, the first is not undone.

$$
\begin{array}{ll}
 & \text{Mach-Port-Insert-Rightp} \\
\equiv & \quad\text{Mach-Port-Insert-Right-Success} \\
 & \vee\ \text{Mach-Port-Insert-Right-Invalid-Task} \\
 & \vee\ \text{Mach-Port-Insert-Right-Invalid-Value} \\
 & \vee\ \text{Mach-Port-Insert-Right-Name-Exists} \\
 & \vee\ \text{Mach-Port-Insert-Right-Invalid-Capability} \\
 & \vee\ \text{Mach-Port-Insert-Right-Urefs-Overflow} \\
 & \vee\ \text{Mach-Port-Insert-Right-Right-Exists} \\
 & \vee\ \text{Mach-Port-Insert-Right-Resource-Shortage}
\end{array}
$$

SPECIFICATION

The success outcome results when the right is extracted from the calling task and inserted by the given name into the target task. The sematics follow closely those of a send followed by a receive.

$$
\begin{array}{ll}
 & \text{Mach-Port-Insert-Right-Success} \\
\equiv & \exists\, p \in \textsc{all-entities}\text{:} \\
 & \quad (\quad \text{Extract-Port-Right}\,(\mathbf{ct},\, \mathbf{n_1},\, \mathbf{instr},\, \mathbf{r},\, p) \\
 & \quad\ \ ;\ \ \text{Insert-Port-Right}\,(\mathbf{t},\, p,\, \mathbf{n_2},\, \mathbf{r}) \\
 & \quad\ \ ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha])
\end{array}
$$

The first set of error outcomes are returned before the name is extracted from the calling task's local name space. An *invalid task* outcome results when the target task is not a task. The *invalid value* outcome results when name **n** is NULLNAME or DEADNAME. The *invalid capability* outcome results when the calling task's arguments $\mathbf{n_1}$, **instr**, and **r** do not make a legal combination. (Unlike explicitly sending a right, the name $\mathbf{n_1}$ cannot designate a dead right.)

The second set of error outcomes are returned after the name is extracted from the calling task's local name space. The *urefs overflow* outcome results when incrementing the reference count for an existing a send right exceeds the maximum. The *name exists* outcome results when $\mathbf{n_2}$ is already in use in $\mathbf{t}$, and the new right cannot be coalesced. The *right exists* outcome results when some other name in $\mathbf{t}$ already has the right.

```
    Mach-Port-Insert-Right-Invalid-Task
$\equiv$ $\Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]$

    Mach-Port-Insert-Right-Invalid-Value
$\equiv$     $\Diamond($     $(\mathbf{n_2} = \textsc{nullname})[\alpha]$
              $\vee$  $(\mathbf{n_2} = \textsc{deadname})[\alpha]$
              $\vee$  $\neg$ Legal-Instr-Right $(\mathbf{instr}, \mathbf{r}))$
      ;  $\Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha]$

    Mach-Port-Insert-Right-Invalid-Capability
$\equiv$     $\Diamond\neg$ Legal-Name-Instr-Right $(\mathbf{ct}, \mathbf{n_1}, \mathbf{instr}, \mathbf{r})$
      ;  $\Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-capability})[\alpha]$

    Mach-Port-Insert-Right-Urefs-Overflow
$\equiv$     $\exists\, p \in \textsc{all-entities}$:
           (     $\Diamond$Extract-Port-Right $(\mathbf{ct}, \mathbf{n_1}, \mathbf{instr}, \mathbf{r}, p)$
             ;       s-right $(\mathbf{t}, \mathbf{n_2})[\alpha]$
                 $\wedge$  (port-right-refcount $(\mathbf{t}, \mathbf{n_2}) = \textsc{max-refcount})[\alpha])$
      ;  $\Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-urefs-overflow})[\alpha]$

    Mach-Port-Insert-Right-Name-Exists
$\equiv$ $\exists\, p \in \textsc{all-entities}$:
      (     $\Diamond$Extract-Port-Right $(\mathbf{ct}, \mathbf{n_1}, \mathbf{instr}, \mathbf{r}, p)$
        ;       Mach-Port-Insert-Right-Send-Once-Name-Exists
            $\vee$  Mach-Port-Insert-Right-Sr-Name-Exists $(p)$
        ;  $\Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-name-exists})[\alpha])$

    Mach-Port-Insert-Right-Send-Once-Name-Exists
$\equiv$ $(\mathbf{r} = \texttt{'send-once})[\alpha]\ ;\ \Diamond$local-namep $(\mathbf{t}, \mathbf{n_2})[\alpha]$

    Mach-Port-Insert-Right-Sr-Name-Exists $(p)$
$\equiv$     $(\mathbf{r} \in \{\texttt{'send}, \texttt{'receive}\})[\alpha]$
      ;  $\Diamond($     local-namep $(\mathbf{t}, \mathbf{n_2})[\alpha]$
            $\wedge$ $\neg$ (     port-right-namep $(\mathbf{t}, \mathbf{n_2})[\alpha]$
                     $\wedge$  (named-port $(\mathbf{t}, \mathbf{n_2}) = p)[\alpha]$
                     $\wedge$  (     port-rights $(\mathbf{t}, \mathbf{n_2})$
                            $\subseteq \{\texttt{'send}, \texttt{'receive}\})[\alpha]))$
    Mach-Port-Insert-Right-Right-Exists
≡ $\exists\, p \in$ ALL-ENTITIES:
     (     Extract-Port-Right ($\mathbf{ct}$, $\mathbf{n_1}$, $\mathbf{instr}$, $\mathbf{r}$, $p$)
       ;  $\Diamond(\mathbf{r} \in \{$'send, 'receive$\})[\alpha]$
       ;  $\exists\, n \in \mathcal{N}$:
            $(\Diamond($     port-right-namep ($\mathbf{t}$, $n$)$[\alpha]$
                 $\wedge$  (named-port ($\mathbf{t}$, $n$) $= p)[\alpha]$
                 $\wedge$  (port-rights ($\mathbf{t}$, $n$) $\subseteq \{$'send, 'receive$\})[\alpha]))$
       ;  $\Diamond\uparrow(\mathbf{rc} =$ 'kern-right-exists$)[\alpha])$

    Mach-Port-Insert-Right-Resource-Shortage
≡ $\Diamond\uparrow(\mathbf{rc} =$ 'kern-resource-shortage$)[\alpha]$
```

3.9 mach_port_mod_refs

DESCRIPTION

Alter the reference count for a right by a given delta. If the count becomes zero, deallocate the right.

PARAMETERS

- **t**. The target task.
- **n**. The local name in **t**'s local name space which is to be affected.
- **type**. The right type for **n** which is to be modified, one of `'send`, `'receive`, `'send-once`, `'port-set`, or `'dead-name`.
- **decr**. A boolean flag which tells whether **n**'s reference count should be decremented or incremented.
- δ. The positive integer amount by which **n**'s reference count is to be modified. In the implementation, δ can be positive or negative, and there is no **decr** flag.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, *invalid right*, *invalid value*, and *urefs overflow.*

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refsp} \\
\equiv & \quad \text{Mach-Port-Mod-Refs-Success} \\
 & \lor\ \text{Mach-Port-Mod-Refs-Invalid-Task} \\
 & \lor\ \text{Mach-Port-Mod-Refs-Invalid-Name} \\
 & \lor\ \text{Mach-Port-Mod-Refs-Invalid-Right} \\
 & \lor\ \text{Mach-Port-Mod-Refs-Invalid-Value} \\
 & \lor\ \text{Mach-Port-Mod-Refs-Urefs-Overflow}
\end{array}
$$

SPECIFICATION

The success outcome occurs when **n** is a local name in **t** of the type specified by **type**. The reference count is incremented or decremented according to **type**, **decr**, and δ. The low-level specifications for modifying rights appear in Section 7.2.

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Success} \\
\equiv & \qquad (\Diamond(\delta = \mathtt{0})[\alpha] \ ;\ \text{Mach-Port-Mod-Refs-Checks}) \\
 & \quad \vee\ (\mathbf{decr}[\alpha] \ ;\ (\delta \neq \mathtt{0})[\alpha] \ ;\ \text{Mach-Port-Mod-Refs-Decrement}) \\
 & \quad \vee\ (\quad (\neg\ \mathbf{decr})[\alpha] \\
 & \qquad\quad ;\ (\delta \neq \mathtt{0})[\alpha] \\
 & \qquad\quad ;\ \text{Mach-Port-Mod-Refs-Increment}) \\
 & ;\ \Diamond{\uparrow}(\mathbf{rc} = \mathtt{'kern\text{-}success})[\alpha]
\end{array}
$$

If δ is zero, the preconditions are checked but there is no transition.

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Checks} \\
\equiv & \quad ((\mathbf{type} = \mathtt{'send})[\alpha] \ ;\ \Diamond\text{s-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]) \\
 & \vee\ ((\mathbf{type} = \mathtt{'receive})[\alpha] \ ;\ \Diamond\text{r-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]) \\
 & \vee\ ((\mathbf{type} = \mathtt{'send\text{-}once})[\alpha] \ ;\ \Diamond\text{so-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]) \\
 & \vee\ ((\mathbf{type} = \mathtt{'port\text{-}set})[\alpha] \ ;\ \Diamond\text{port-set-namep}\,(\mathbf{t},\ \mathbf{n})[\alpha]) \\
 & \vee\ ((\mathbf{type} = \mathtt{'dead\text{-}name})[\alpha] \ ;\ \Diamond\text{dead-right-namep}\,(\mathbf{t},\ \mathbf{n})[\alpha])
\end{array}
$$

Only send rights or dead rights may have their reference counts incremented. The new value must be less than or equal to the maximum allowed reference count.

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Increment} \\
\equiv & \quad (\quad (\mathbf{type} = \mathtt{'send})[\alpha] \\
 & \qquad ;\ \exists\, p \in \textsc{all-entities}\text{:}\ \ (\Diamond\text{Coalesce-Send-Right}\,(\mathbf{t},\ p,\ \mathbf{n},\ \delta))) \\
 & \vee\ ((\mathbf{type} = \mathtt{'dead\text{-}name})[\alpha] \ ;\ \Diamond\text{Coalesce-Dead-Right}\,(\mathbf{t},\ \mathbf{n},\ \delta))
\end{array}
$$

If a reference count is decremented to zero, the right is deleted. Send rights and dead rights can have reference counts greater than or equal to one. For this kernel call, receive rights, send-once rights, and port sets are considered to have a reference count of exactly one.

When a receive right is deleted, the port is terminated.

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Decrement} \\
\equiv & \quad (\Diamond(\mathbf{type} = \texttt{'send})[\alpha] \;;\; \Diamond\text{Mach-Port-Mod-Refs-Decrement-Sendp}) \\
 & \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'receive})[\alpha] \\
 & \qquad ;\ \Diamond\text{Mach-Port-Mod-Refs-Decrement-Receivep}) \\
 & \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'send-once})[\alpha] \\
 & \qquad ;\ \Diamond\text{Mach-Port-Mod-Refs-Decrement-Send-Oncep}) \\
 & \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'port-set})[\alpha] \\
 & \qquad ;\ \Diamond\text{Mach-Port-Mod-Refs-Decrement-Port-Setp}) \\
 & \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'dead-name})[\alpha] \\
 & \qquad ;\ \Diamond\text{Mach-Port-Mod-Refs-Decrement-Dead-Namep})
\end{array}
$$

$$\text{Mach-Port-Mod-Refs-Decrement-Sendp} \equiv \text{Deallocate-Send-Right}\,(\mathbf{t},\, \mathbf{n},\, \delta)$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Decrement-Receivep} \\
\equiv & \exists\, p \in \text{ENTITIES}: \\
 & \quad (\quad \Diamond(\delta = \texttt{1})[\alpha] \\
 & \qquad ;\ \Diamond(\text{r-right}\,(\mathbf{t},\, \mathbf{n})[\alpha] \wedge (\text{named-port}\,(\mathbf{t},\, \mathbf{n}) = p)[\alpha]) \\
 & \qquad ;\ \text{Deallocate-Receive-Right}\,(\mathbf{t},\, \mathbf{n}) \\
 & \qquad ;\ \text{Terminate-Port}\,(p))
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Decrement-Send-Oncep} \\
\equiv & \Diamond(\delta = \texttt{1})[\alpha] \;;\; \Diamond\text{Remove-Send-Once-Right}\,(\mathbf{t},\, \mathbf{n})
\end{array}
$$

$$\text{Mach-Port-Mod-Refs-Decrement-Dead-Namep} \equiv \text{Deallocate-Dead-Right}\,(\mathbf{t},\, \mathbf{n},\, \delta)$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Mod-Refs-Decrement-Port-Setp} \\
\equiv & \Diamond(\delta = \texttt{1})[\alpha] \;;\; \Diamond\text{Remove-Port-Set-Name}\,(\mathbf{t},\, \mathbf{n})
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The *invalid right* outcome results when **n** is a local name in **t**, but not the expected type of right.

The *invalid value* outcome results when **type** is not a legal value, or when δ is out of the range of the reference count of **n**. For **type** $\in${`'receive`, `'send-once`, `'port-set`}, one is allowed to either supply δ =0 or decrement by one. For `'send-once` or `'dead-name`, one cannot decrement a reference count below zero.

The *urefs overflow* outcome results when incrementing the reference count by the designated δ would be larger than the allowed maximum.

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Task}\\
\equiv\ &\Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Name}\\
\equiv\ &\Diamond(\neg\ \text{local-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Right}\\
\equiv\ &\qquad\quad (\Diamond(\mathbf{type} = \texttt{'send})[\alpha]\ ;\ (\neg\ \text{s-right}\,(\mathbf{t},\ \mathbf{n}))[\alpha])\\
&\qquad \vee\ (\Diamond(\mathbf{type} = \texttt{'receive})[\alpha]\ ;\ (\neg\ \text{r-right}\,(\mathbf{t},\ \mathbf{n}))[\alpha])\\
&\qquad \vee\ (\Diamond(\mathbf{type} = \texttt{'send-once})[\alpha]\ ;\ (\neg\ \text{so-right}\,(\mathbf{t},\ \mathbf{n}))[\alpha])\\
&\qquad \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'port-set})[\alpha]\\
&\qquad\qquad ;\ \ (\neg\ \text{port-set-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha])\\
&\qquad \vee\ (\quad \Diamond(\mathbf{type} = \texttt{'dead-name})[\alpha]\\
&\qquad\qquad ;\ \ (\neg\ \text{dead-right-namep}\,(\mathbf{t},\ \mathbf{n}))[\alpha])\\
&\ \ ;\ \ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Value}\\
\equiv\ &\qquad\quad \Diamond(\mathbf{type} \notin \{\texttt{'send},\ \texttt{'receive},\ \texttt{'send-once},\ \texttt{'port-set},\\
&\qquad\qquad\qquad\qquad \texttt{'dead-name}\})[\alpha]\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Send-Value}\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Send-Receive-Value}\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Receive-Value}\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Send-Once-Value}\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Dead-Name-Value}\\
&\qquad \vee\ \text{Mach-Port-Mod-Refs-Invalid-Port-Set-Value}\\
&\ \ ;\ \ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha]
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Send-Value}\\
\equiv\ &\quad \Diamond((\mathbf{type} = \texttt{'send})[\alpha] \wedge \mathbf{decr}[\alpha])\\
&\ \ ;\ \ \Diamond(\quad \text{s-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]\\
&\qquad\ \wedge\ (\neg\ \text{r-right}\,(\mathbf{t},\ \mathbf{n}))[\alpha]\\
&\qquad\ \wedge\ (\text{port-right-refcount}\,(\mathbf{t},\ \mathbf{n}) < \delta)[\alpha])
\end{aligned}
$$

$$
\begin{aligned}
&\text{Mach-Port-Mod-Refs-Invalid-Send-Receive-Value}\\
\equiv\ &\quad \Diamond((\mathbf{type} = \texttt{'send})[\alpha] \wedge \mathbf{decr}[\alpha])\\
&\ \ ;\ \ \Diamond(\quad \text{s-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]\\
&\qquad\ \wedge\ \text{r-right}\,(\mathbf{t},\ \mathbf{n})[\alpha]\\
&\qquad\ \wedge\ (\text{port-right-refcount}\,(\mathbf{t},\ \mathbf{n}) - \texttt{1} < \delta)[\alpha])
\end{aligned}
$$

Mach-Port-Mod-Refs-Invalid-Receive-Value

$$\begin{aligned}\equiv \Diamond(\quad & (\textbf{type} = \texttt{'receive})[\alpha] \\ \wedge\ & \neg\,(\delta = \texttt{0})[\alpha] \\ \wedge\ & ((\neg\,\textbf{decr})[\alpha] \vee (\delta \neq \texttt{1})[\alpha]))\end{aligned}$$

Mach-Port-Mod-Refs-Invalid-Send-Once-Value

$$\begin{aligned}\equiv \Diamond(\quad & (\textbf{type} = \texttt{'send-once})[\alpha] \\ \wedge\ & \neg\,(\delta = \texttt{0})[\alpha] \\ \wedge\ & ((\neg\,\textbf{decr})[\alpha] \vee (\delta \neq \texttt{1})[\alpha]))\end{aligned}$$

Mach-Port-Mod-Refs-Invalid-Dead-Name-Value

$$\begin{aligned}\equiv (\quad & \Diamond(\textbf{type} = \texttt{'dead-name})[\alpha] \\ ;\ & \Diamond\textbf{decr}[\alpha] \\ ;\ & \exists\, \texttt{1} \leq i < \textsc{max-refcount}: \\ & \quad (\Diamond\text{dead-right-rel}\,(\textbf{t},\, \textbf{n},\, i)[\alpha] \;;\; \Diamond(i < \delta)[\alpha]))\end{aligned}$$

Mach-Port-Mod-Refs-Invalid-Port-Set-Value

$$\begin{aligned}\equiv \Diamond(\quad & (\textbf{type} = \texttt{'port-set})[\alpha] \\ \wedge\ & \neg\,(\delta = \texttt{0})[\alpha] \\ \wedge\ & ((\neg\,\textbf{decr})[\alpha] \vee (\delta \neq \texttt{1})[\alpha]))\end{aligned}$$

Mach-Port-Mod-Refs-Urefs-Overflow

$$\begin{aligned}\equiv \quad & \quad (\quad \Diamond(\delta > \texttt{0} \wedge (\textbf{type} = \texttt{'send}))[\alpha] \\ & \quad\quad ;\ \text{Mach-Port-Mod-Refs-Overflow-Sendp}) \\ & \vee\ (\quad \Diamond(\delta > \texttt{0} \wedge (\textbf{type} = \texttt{'dead-name}))[\alpha] \\ & \quad\quad ;\ \text{Mach-Port-Mod-Refs-Overflow-Dead-Namep}) \\ ;\ & \Diamond\!\uparrow\!(\textbf{rc} = \texttt{'kern-urefs-overflow})[\alpha]\end{aligned}$$

Mach-Port-Mod-Refs-Overflow-Sendp

$$\begin{aligned}\equiv \quad & \Diamond(\quad \text{s-right}\,(\textbf{t},\, \textbf{n})[\alpha] \\ & \quad \wedge\ (\neg\,\text{r-right}\,(\textbf{t},\, \textbf{n}))[\alpha] \\ & \quad \wedge\ (\text{port-right-refcount}\,(\textbf{t},\, \textbf{n}) + \delta > \textsc{max-refcount})[\alpha]) \\ \vee \quad & \text{s-right}\,(\textbf{t},\, \textbf{n})[\alpha] \\ & \wedge\ \text{r-right}\,(\textbf{t},\, \textbf{n})[\alpha] \\ & \wedge\ ((\text{port-right-refcount}\,(\textbf{t},\, \textbf{n}) - \texttt{1}) + \delta > \textsc{max-refcount})[\alpha]\end{aligned}$$

Mach-Port-Mod-Refs-Overflow-Dead-Namep

$$\begin{aligned}\equiv\ & \exists\, \texttt{1} \leq i < \textsc{max-refcount}: \\ & \quad (\Diamond(\text{dead-right-rel}\,(\textbf{t},\, \textbf{n},\, i)[\alpha] \wedge (i + \delta > \textsc{max-refcount})[\alpha]))\end{aligned}$$

3.10 mach_port_move_member

DESCRIPTION

Move a receive right into or out of a port set.

PARAMETERS

- **t**. The target task.
- $\mathbf{n_1}$. **t**'s local name for a receive right.
- $\mathbf{n_2}$. Either **t**'s local name for a port set, or NULLNAME.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, *invalid right*, *invalid value*, and *not in set*.

$$\begin{array}{ll} & \text{Mach-Port-Move-Memberp} \\ \equiv & \quad \text{Mach-Port-Move-Member-Success} \\ & \vee\ \text{Mach-Port-Move-Member-Invalid-Task} \\ & \vee\ \text{Mach-Port-Move-Member-Invalid-Name} \\ & \vee\ \text{Mach-Port-Move-Member-Invalid-Right} \\ & \vee\ \text{Mach-Port-Move-Member-Not-In-Setp} \end{array}$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the receive right is moved either into or out of its port set, as indicated by the $\mathbf{n_2}$ argument.

$$\begin{array}{ll} & \text{Mach-Port-Move-Member-Success} \\ \equiv & \quad \Diamond\text{r-right}\,(\mathbf{t},\,\mathbf{n_1})[\alpha] \\ & ;\quad (\quad \Diamond(\mathbf{n_2} = \text{NULLNAME})[\alpha] \\ & \qquad\quad ;\ \text{Mach-Port-Move-Member-Out-Of}) \\ & \quad\ \vee\ (\Diamond(\mathbf{n_2} \neq \text{NULLNAME})[\alpha]\ ;\ \text{Mach-Port-Move-Member-Into}) \\ & ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

Mach-Port-Move-Member-Out-Of

$\equiv \Diamond\text{in-port-set}\,(\mathbf{t},\, \mathbf{n_1})[\alpha]\ ;\ \Diamond\neg\ \text{in-port-set}\,(\mathbf{t},\, \mathbf{n_1})[\alpha]$

Mach-Port-Move-Member-Into

$$\begin{aligned}
\equiv\ & \Diamond\neg\ \text{in-port-set}\,(\mathbf{t},\, \mathbf{n_1})[\alpha] \\
;\ & \Diamond\text{port-set-namep}\,(\mathbf{t},\, \mathbf{n_2})[\alpha] \\
;\ & \Diamond(\quad \text{port-set-namep}\,(\mathbf{t},\, \mathbf{n_2})[\alpha] \\
& \quad\wedge\ (\mathbf{n_1} \in \text{port-set}\,(\mathbf{t},\, \mathbf{n_2}))[\alpha])
\end{aligned}$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name n is not a local name in the target task. The *invalid right* outcome results when n is a local name in t, but not the expected type of right. The *not in set* outcome results when we intend to remove $\mathbf{n_1}$ from its port set, but it is not a member of a port set.

Mach-Port-Move-Member-Invalid-Task

$\equiv \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]$

Mach-Port-Move-Member-Invalid-Name

$$\begin{aligned}
\equiv\ & \Diamond\text{taskp}\,(\mathbf{t})[\alpha] \\
;\ & \quad \Diamond(\neg\ \text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}))[\alpha] \\
& \vee\ \Diamond(\mathbf{n_2} \neq \text{NULLNAME} \wedge \neg\ \text{local-namep}\,(\mathbf{t},\, \mathbf{n_2}))[\alpha] \\
;\ & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{aligned}$$

Mach-Port-Move-Member-Invalid-Right

$$\begin{aligned}
\equiv\ & \quad \Diamond(\text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}) \wedge \neg\ \text{r-right}\,(\mathbf{t},\, \mathbf{n_1}))[\alpha] \\
& \vee\ \Diamond(\quad (\mathbf{n_2} \neq \text{NULLNAME})[\alpha] \\
& \qquad \wedge\ \text{local-namep}\,(\mathbf{t},\, \mathbf{n_2})[\alpha] \\
& \qquad \wedge\ (\neg\ \text{port-set-namep}\,(\mathbf{t},\, \mathbf{n_2}))[\alpha]) \\
;\ & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{aligned}$$

Mach-Port-Move-Member-Not-In-Setp

$$\begin{aligned}
\equiv\ & \Diamond(\mathbf{n_2} = \text{NULLNAME})[\alpha] \\
;\ & \Diamond(\text{r-right}\,(\mathbf{t},\, \mathbf{n_1})[\alpha] \wedge \neg\ \text{in-port-set}\,(\mathbf{t},\, \mathbf{n_1})[\alpha]) \\
;\ & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-not-in-set})[\alpha]
\end{aligned}$$

3.11 mach_port_names

DESCRIPTION

Return information about a task's local name space.

PARAMETERS

- **t**. The target task.
- **names**. [out] A snapshot of all local names in **t**'s local name space.
- **types**. [out] The types of the elements of **names**.

OUTCOMES

The possible outcomes are *success*, *invalid task*, and *resource shortage.*

```
   Mach-Port-Namesp
≡     Mach-Port-Names-Success
    ∨ Mach-Port-Names-Invalid-Task
    ∨ Mach-Port-Names-Resource-Shortage
```

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and **names** and **types** contain a snapshot of **t**'s name space. The names are returned in no particular order.

The type information for each name is a set containing elements from the following set:

`'send`: The name denotes a send right.

`'receive`: The name denotes a receive right.

`'send-once`: The name denotes a send-once right.

`'port-set`: The name denotes a port set.

`'dead-name`: The name is a dead name.

`'dnrequest`: A dead-name request has been registered for the right.

$$
\begin{array}{ll}
 & \text{Mach-Port-Names-Success} \\
\equiv & \quad \Diamond\forall\, n \in \mathcal{N}: \\
 & \qquad (\quad \text{local-namep}\,(\mathbf{t},\, n)[\alpha] \\
 & \qquad\ \to \exists\, 0 \le i < |\mathcal{N}|: \\
 & \qquad\qquad (\quad \Diamond{\uparrow}(n = \mathbf{names}_i)[\alpha] \\
 & \qquad\qquad\ \wedge\ (\text{r-right}\,(\mathbf{t},\, n)[\alpha] \to \Diamond{\uparrow}\texttt{'receive} \in \mathbf{types}_i[\alpha]) \\
 & \qquad\qquad\ \wedge\ (\text{s-right}\,(\mathbf{t},\, n)[\alpha] \to \Diamond{\uparrow}\texttt{'send} \in \mathbf{types}_i[\alpha]) \\
 & \qquad\qquad\ \wedge\ (\quad \text{so-right}\,(\mathbf{t},\, n)[\alpha] \\
 & \qquad\qquad\qquad \to \Diamond{\uparrow}\texttt{'send-once} \in \mathbf{types}_i[\alpha]) \\
 & \qquad\qquad\ \wedge\ (\quad \text{dead-right-namep}\,(\mathbf{t},\, n)[\alpha] \\
 & \qquad\qquad\qquad \to \Diamond{\uparrow}\texttt{'dead-name} \in \mathbf{types}_i[\alpha]) \\
 & \qquad\qquad\ \wedge\ (\quad \text{exists-dn-notification-port}\,(\mathbf{t},\, n)[\alpha] \\
 & \qquad\qquad\qquad \to \Diamond{\uparrow}\texttt{'dnrequest} \in \mathbf{types}_i[\alpha]))) \\
 & \ ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

An *invalid task* outcome results when the target task is not a task.

$$
\begin{array}{ll}
 & \text{Mach-Port-Names-Invalid-Task} \\
\equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Names-Resource-Shortage} \\
\equiv & \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha]
\end{array}
$$

3.12 mach_port_rename

DESCRIPTION

Change a local port name.

PARAMETERS

- $\mathbf{t}$. The target task.
- $\mathbf{n_1}$. The local name in $\mathbf{t}$ which is to be changed.
- $\mathbf{n_2}$. The new name.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid right*, *invalid name*, *invalid value*, *invalid right*, *name exists*, and *resource shortage*.

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Mach-Port-Renamep}} \\ \equiv & & \text{Mach-Port-Rename-Success} \\ & \lor & \text{Mach-Port-Rename-Invalid-Task} \\ & \lor & \text{Mach-Port-Rename-Invalid-Name} \\ & \lor & \text{Mach-Port-Rename-Invalid-Value} \\ & \lor & \text{Mach-Port-Rename-Name-Exists} \\ & \lor & \text{Mach-Port-Rename-Resource-Shortage} \end{array}$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the rename is accomplished.

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Mach-Port-Rename-Success}} \\ \equiv & & \Diamond(\quad ((\mathbf{n_2} \in \mathcal{N}) \land \mathbf{n_2} \neq \text{NULLNAME})[\alpha] \\ & & \quad\quad \land\ \text{taskp}\,(\mathbf{t})[\alpha] \\ & & \quad\quad \land\ (\quad \text{Mach-Port-Rename-Port-Rightp} \\ & & \quad\quad\quad\quad \lor\ \text{Mach-Port-Rename-Port-Set-Namep} \\ & & \quad\quad\quad\quad \lor\ \text{Mach-Port-Rename-Dead-Namep})) \\ & ; & \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

Mach-Port-Rename-Port-Rightp

$$\begin{array}{l} \equiv \exists\, p \in \textsc{entities},\, R \in \textsc{all-rsets},\, 1 \leq i < \textsc{max-refcount}: \\ \quad (\quad \Diamond\text{port-right-rel}\,(\mathbf{t},\, \mathbf{n_1},\, p,\, R,\, i)[\alpha] \\ \quad\;\; ;\; \Diamond\!\uparrow \quad \neg\, \text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}) \\ \qquad\qquad \wedge\; \text{port-right-rel}\,(\mathbf{t},\, \mathbf{n_2},\, p,\, R,\, i)[\alpha]) \end{array}$$

Mach-Port-Rename-Port-Set-Namep

$$\begin{array}{l} \equiv \exists\, N \in \textsc{all-nsets}: \\ \quad (\quad \Diamond\text{port-set-rel}\,(\mathbf{t},\, \mathbf{n_1},\, N)[\alpha] \\ \quad\;\; ;\; \Diamond\!\uparrow\!\neg\, \text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}) \wedge \text{port-set-rel}\,(\mathbf{t},\, \mathbf{n_2},\, N)[\alpha]) \end{array}$$

Mach-Port-Rename-Dead-Namep

$$\begin{array}{l} \equiv \exists\, 0 \leq i < \textsc{max-refcount}: \\ \quad (\quad \Diamond\text{dead-right-rel}\,(\mathbf{t},\, \mathbf{n_1},\, i)[\alpha] \\ \quad\;\; ;\; \Diamond\!\uparrow \quad \neg\, \text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}) \\ \qquad\qquad \wedge\; \text{dead-right-rel}\,(\mathbf{t},\, \mathbf{n_2},\, i)[\alpha]) \end{array}$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name $\mathbf{n_1}$ is not a local name in the target task. The *invalid value* outcome results when $\mathbf{n_2}$ is not a valid name. The *name exists* outcome results when $\mathbf{n_2}$ is already a local name in $\mathbf{t}$.

Mach-Port-Rename-Invalid-Task

$$\equiv \Diamond(\neg\, \text{taskp}\,(\mathbf{t}))[\alpha] \;;\; \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]$$

Mach-Port-Rename-Invalid-Name

$$\begin{array}{l} \equiv \quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\, \text{local-namep}\,(\mathbf{t},\, \mathbf{n_1}))[\alpha] \\ \quad ;\; \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha] \end{array}$$

Mach-Port-Rename-Invalid-Value

$$\begin{array}{l} \equiv \quad \Diamond(\quad \neg\, (\mathbf{n_2} \in \mathcal{N}) \\ \qquad\quad \vee\; (\mathbf{n_2} = \textsc{nullname}) \\ \qquad\quad \vee\; (\mathbf{n_2} = \textsc{deadname}))[\alpha] \\ \quad ;\; \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha] \end{array}$$

Mach-Port-Rename-Name-Exists

$$\equiv \Diamond\text{local-namep}\,(\mathbf{t},\, \mathbf{n_2})[\alpha] \;;\; \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-name-exists})[\alpha]$$

Mach-Port-Rename-Resource-Shortage

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

3.13 mach_port_request_notification

DESCRIPTION

Request a notification of a port event.

PARAMETERS

- **t**. The target task.
- $\mathbf{n_1}$. The local name in **t**'s name space for which the notification should be registered.
- **variant**. The variant of notification requested, one of `'port-destroyed`, `'dead-name`, or `'no-senders`. Port-destryed notifications are not a part of our specification.
- **sync**. Some variants use this to overcome race conditions. We do not model these cases.
- $\mathbf{n_2}$. The local name to which the notification will be sent. It can be NULLNAME to cancel a previous request.
- **instr**. Instruction for creating a send-once right from $\mathbf{n_2}$, either `'make` or `'move`.
- $\mathbf{n_3}$. [out] The previous notification send-once right, if any. We do not model the effects on this argument.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid value*, *invalid name*, *invalid right*, *invalid capability*, *invalid argument*, *urefs overflow*, and *resource shortage*.

$$
\begin{array}{ll}
& \text{Mach-Port-Request-Notificationp} \\
\equiv & \quad \text{Mach-Port-Request-Notification-Success} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Task} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Value} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Name} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Right} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Capability} \\
& \vee\ \text{Mach-Port-Request-Notification-Invalid-Argumentp} \\
& \vee\ \text{Mach-Port-Request-Notification-Urefs-Overflow} \\
& \vee\ \text{Mach-Port-Request-Notification-Resource-Shortage}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the appropriate notification is either registered or canceled.

$$
\begin{array}{ll}
& \text{Mach-Port-Request-Notification-Success} \\
\equiv & \qquad \Diamond(\mathbf{variant} = \texttt{'port-destroyed})[\alpha] \\
& \quad \vee\ \text{Mach-Port-Request-Notification-Dead-Name} \\
& \quad \vee\ \text{Mach-Port-Request-Notification-No-Senders} \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'success})[\alpha]
\end{array}
$$

For a dead name notification, destruction of the port denoted by name n_1 in task t causes a message containing n_1 to be sent to port p. (When the name of a task's capability on a port is changed (via `mach_port_rename`), the dead-name notification is modified to reflect the new name.)

$$
\begin{array}{ll}
& \text{Mach-Port-Request-Notification-Dead-Name} \\
\equiv & \quad \Diamond(\mathbf{variant} = \texttt{'dead-name})[\alpha] \\
& ;\quad \text{Mach-Port-Request-Notification-Dead-Name-Register} \\
& \quad \vee\ \text{Mach-Port-Request-Notification-Dead-Name-Cancel}
\end{array}
$$

$$
\begin{array}{ll}
& \text{Mach-Port-Request-Notification-Dead-Name-Register} \\
\equiv & \forall\, p \in \text{ALL-ENTITIES:} \\
& \quad (\quad \Diamond\text{port-right-namep}\,(\mathbf{t},\, \mathbf{n_1})[\alpha] \\
& \quad\ ;\ \text{Extract-Port-Right}\,(\mathbf{t},\, \mathbf{n_2},\, \mathbf{instr},\, \texttt{'send-once},\, p) \\
& \quad\ ;\ \Diamond\uparrow\text{dn-notification-rel}\,(p,\, \mathbf{t},\, \mathbf{n_1})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Request-Notification-Dead-Name-Cancel} \\
\equiv & \quad \Diamond \text{port-right-namep}\,(\mathbf{t},\ \mathbf{n_1})[\alpha] \\
; & \Diamond(\mathbf{n_2} = \text{NULLNAME})[\alpha] \\
; & \Diamond(\neg\ \text{exists-dn-notification-port}\,(\mathbf{t},\ \mathbf{n_1}))[\alpha]
\end{array}
$$

When `'(equal variant 'no-senders)`, n_1 must specify a receive right. If n_2 is not null, and the receive right's make-send count is greater than or equal to the sync value, and it has no extant send rights, than an immediate no-senders notification is generated. Otherwise the notification is generated when the receive right next loses its last extant send right.

$$
\begin{array}{ll}
 & \text{Mach-Port-Request-Notification-No-Senders} \\
\equiv & \quad \Diamond(\mathbf{variant} = \texttt{'no-senders})[\alpha] \\
; & \quad\quad \text{Mach-Port-Request-Notification-No-Senders-Register} \\
 & \quad \vee\ \text{Mach-Port-Request-Notification-No-Senders-Cancel}
\end{array}
$$

$$
\begin{array}{l}
\text{Mach-Port-Request-Notification-No-Senders-Register} \\
\equiv \exists\, p_1 \in \text{ALL-ENTITIES},\ p_2 \in \text{ALL-ENTITIES}: \\
\quad (\quad \Diamond(\quad \text{r-right}\,(\mathbf{t},\ \mathbf{n_1})[\alpha] \\
\quad\quad\quad\quad \wedge\ (\text{named-port}\,(\mathbf{t},\ \mathbf{n_1}) = p_1)[\alpha]) \\
\quad\quad ;\ \text{Extract-Port-Right}\,(\mathbf{t},\ \mathbf{n_2},\ \mathbf{instr},\ \texttt{'send-once},\ p_2) \\
\quad\quad ;\ \Diamond{\uparrow}\text{ns-notification-rel}\,(p_2,\ p_1)[\alpha])
\end{array}
$$

$$
\begin{array}{l}
\text{Mach-Port-Request-Notification-No-Senders-Cancel} \\
\equiv \exists\, p \in \text{ALL-ENTITIES}: \\
\quad (\quad \Diamond(\quad \text{r-right}\,(\mathbf{t},\ \mathbf{n_1})[\alpha] \\
\quad\quad\quad\quad \wedge\ (\text{named-port}\,(\mathbf{t},\ \mathbf{n_1}) = p)[\alpha]) \\
\quad\quad ;\ \Diamond(\mathbf{n_2} = \text{NULLNAME})[\alpha] \\
\quad\quad ;\ \Diamond{\downarrow}\text{exists-ns-notification-port}\,(p)[\alpha])
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid value* outcome results when **variant** was not an expected value. The *invalid name* outcome results when name $\mathbf{n_1}$ is not a local name in the target task.

The *invalid right* outcome results when $\mathbf{n_1}$ is a local name in $\mathbf{t}$, but not the expected type of right. The *invalid capability* outcome results when $\mathbf{n_2}$ was not a valid right. The *invalid argument* outcome results when $\mathbf{n_1}$ denotes a dead name, but **sync** is zero or $\mathbf{n_2} = \text{NULLNAME}$. The *urefs overflow* outcome results when $\mathbf{n_1}$ denotes a dead name, but generating an immediate dead-name notification would overflow the name's user-reference count.

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Task}\\
\equiv\ &\Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Value}\\
\equiv\ &\quad \Diamond(\mathbf{variant} \notin \{\texttt{'port-destroyed}, \texttt{'dead-name}, \texttt{'no-senders}\})[\alpha]\\
&\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Name}\\
\equiv\ &\quad \Diamond(\neg\ \text{local-namep}\,(\mathbf{t}, \mathbf{n_1}))[\alpha]\\
&\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Right}\\
\equiv\ &\quad \Diamond(\quad \text{local-namep}\,(\mathbf{t}, \mathbf{n_1})[\alpha]\\
&\qquad\quad \wedge\ (\quad (\mathbf{variant} = \texttt{'port-destroyed})[\alpha]\\
&\qquad\qquad\quad \rightarrow (\neg\ \text{r-right}\,(\mathbf{t}, \mathbf{n_1}))[\alpha])\\
&\qquad\quad \wedge\ (\quad (\mathbf{variant} = \texttt{'dead-name})[\alpha]\\
&\qquad\qquad\quad \rightarrow \quad (\neg\ \text{s-right}\,(\mathbf{t}, \mathbf{n_1}))[\alpha]\\
&\qquad\qquad\qquad \wedge\ (\neg\ \text{r-right}\,(\mathbf{t}, \mathbf{n_1}))[\alpha]\\
&\qquad\qquad\qquad \wedge\ (\neg\ \text{so-right}\,(\mathbf{t}, \mathbf{n_1}))[\alpha])\\
&\qquad\quad \wedge\ (\quad (\mathbf{variant} = \texttt{'no-senders})[\alpha]\\
&\qquad\qquad\quad \rightarrow (\neg\ \text{r-right}\,(\mathbf{t}, \mathbf{n_1}))[\alpha]))\\
&\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Capability}\\
\equiv\ &\quad \Diamond\neg\ \text{Legal-Name-Instr-Right}\,(\mathbf{t}, \mathbf{n_2}, \mathbf{instr}, \texttt{'send-once})\\
&\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-capability})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Invalid-Argumentp}\\
\equiv\ &\qquad \Diamond\text{dead-right-namep}\,(\mathbf{t}, \mathbf{n_1})[\alpha]\\
&\quad \wedge\ (\Diamond(\mathbf{n_2} = \text{NULLNAME})[\alpha] \vee \Diamond(\mathbf{sync} = \texttt{0})[\alpha])\\
&\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\text{Mach-Port-Request-Notification-Urefs-Overflow}\\
\equiv\ &\Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-urefs-overflow})[\alpha]
\end{aligned}$$

$$\begin{array}{ll} & \text{Mach-Port-Request-Notification-Resource-Shortage} \\ \equiv & \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha] \end{array}$$

3.14 mach_port_set_qlimit

DESCRIPTION

Set the queue limit for a port.

PARAMETERS

- **t**. The target task.
- **n**. The name of a receive right in **t**.
- **i**. The number of messages which may be queued to this port without causing the sender to block.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid name*, *invalid right*, and *invalid value.*

$$
\begin{array}{ll}
 & \text{Mach-Port-Set-Qlimitp} \\
\equiv & \quad \text{Mach-Port-Set-Qlimit-Success} \\
 & \lor\ \text{Mach-Port-Set-Qlimit-Invalid-Task} \\
 & \lor\ \text{Mach-Port-Set-Qlimit-Invalid-Name} \\
 & \lor\ \text{Mach-Port-Set-Qlimit-Invalid-Right} \\
 & \lor\ \text{Mach-Port-Set-Qlimit-Invalid-Value}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the new limit is set[4]. Note that it is *not* a kernel invariant that the length of a port's message queue is less than its queue limit. The limit may be set to less than the current length without causing any change to the queue[5].

[4]We use the notation $\Diamond p[\alpha]$ (occurs) rather than $\Diamond\uparrow p[\alpha]$ (asserted) because it is legitimate to set the value to the one it currently has.

[5]If the queue limit is increased, blocked senders may be awakened. We do not model this behavior explicitly.

Mach-Port-Set-Qlimit-Success

$$
\begin{array}{ll}
\equiv & \exists\, p \in \textsc{all-entities}: \\
& \quad (\quad \Diamond(\text{r-right}\,(\mathbf{t}, \mathbf{n}) \wedge (p = \text{named-port}\,(\mathbf{t}, \mathbf{n})))[\alpha] \\
& \qquad ;\ \ \Diamond\text{message-qlimit-rel}\,(p, \mathbf{i})[\alpha] \\
& \qquad ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha])
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name **n** is not a local name in the target task. The *invalid right* outcome results when **n** is a local name in **t**, but not a receive right. The *invalid value* outcome results when the new queue limit is greater than the maximum allowed.

Mach-Port-Set-Qlimit-Invalid-Task

$$
\equiv\ \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
$$

Mach-Port-Set-Qlimit-Invalid-Name

$$
\begin{array}{ll}
\equiv & \quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t}, \mathbf{n}))[\alpha] \\
& ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{array}
$$

Mach-Port-Set-Qlimit-Invalid-Right

$$
\begin{array}{ll}
\equiv & \quad \Diamond(\text{local-namep}\,(\mathbf{t}, \mathbf{n}) \wedge \neg\ \text{r-right}\,(\mathbf{t}, \mathbf{n}))[\alpha] \\
& ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-right})[\alpha]
\end{array}
$$

Mach-Port-Set-Qlimit-Invalid-Value

$$
\equiv\ \Diamond(\Diamond(\mathbf{i} > \textsc{max-qlimit})[\alpha])\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-value})[\alpha]
$$

3.15 mach_port_type

DESCRIPTION

Return information about a local name.

PARAMETERS

- **t**. The target task.
- **n**. The local name of interest.
- **types**. [out] A subset of the set {`'send`, `'receive`, `'send-once`, `'port-set`, `'dead-name`}.
- **flags**. [out] A subset of the set {`'dnrequest`, `'marequest`, `'compat`}[6].

OUTCOMES

The possible outcomes are *success*, *invalid task*, and *invalid name.*

$$
\begin{array}{ll}
 & \text{Mach-Port-Typep} \\
\equiv & \quad \text{Mach-Port-Type-Success} \\
 & \vee\ \text{Mach-Port-Type-Invalid-Task} \\
 & \vee\ \text{Mach-Port-Type-Invalid-Name}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and **types** and **flags** contain the information about **n**.

$$
\begin{array}{lll}
 & \text{Mach-Port-Type-Success} & \\
\equiv & \Diamond(& \quad \text{Mach-Port-Port-Name-Typep} \\
 & & \vee\ \text{Mach-Port-Dead-Name-Typep} \\
 & & \vee\ \text{Mach-Port-Port-Set-Typep}) \\
 & ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha] &
\end{array}
$$

[6]The `'marequest` flag indicates that a msg-accepted request for the right is pending. The `'compat` flag indicates that this port was created in Mach 2.5 compatibility mode. We do not model these values.

$$
\begin{array}{ll}
 & \text{Mach-Port-Port-Name-Typep} \\
\equiv & \Diamond(\quad \text{port-right-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha] \\
 & \quad\wedge\ (\mathbf{types} = \text{port-rights}\,(\mathbf{t},\,\mathbf{n}))[\alpha] \\
 & \quad\wedge\ (\text{exists-dn-notification-port}\,(\mathbf{t},\,\mathbf{n}) \leftrightarrow \texttt{'dnrequest} \in \mathbf{flags})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Dead-Name-Typep} \\
\equiv & \Diamond(\quad \text{dead-right-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha] \\
 & \quad\wedge\ (\mathbf{types} = \{\texttt{'dead-name}\})[\alpha] \\
 & \quad\wedge\ (\mathbf{flags} = \emptyset)[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Port-Set-Typep} \\
\equiv & \Diamond(\quad \text{port-set-namep}\,(\mathbf{t},\,\mathbf{n})[\alpha] \\
 & \quad\wedge\ (\mathbf{types} = \{\texttt{'port-set}\})[\alpha] \\
 & \quad\wedge\ (\mathbf{flags} = \emptyset)[\alpha])
\end{array}
$$

An *invalid task* outcome results when the target task is not a task. The *invalid name* outcome results when name $\mathbf{n_1}$ is not a local name in the target task.

$$
\begin{array}{ll}
 & \text{Mach-Port-Type-Invalid-Task} \\
\equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-task})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Mach-Port-Type-Invalid-Name} \\
\equiv & \quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{local-namep}\,(\mathbf{t},\,\mathbf{n}))[\alpha] \\
 & ;\ \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-name})[\alpha]
\end{array}
$$

3.16 mach_reply_port

DESCRIPTION

Create a reply port.

PARAMETERS

- **t**. The calling task. This is implicit in the implementation.
- **n**. [out] Either the new local name, or NULLNAME to indicate failure.

SPECIFICATION

This is an optimized version of mach_port_allocate, with a different interface. The task argument and the return code do not appear in the interface. For the former, the current task is supplied implicitly. When there is a non-success return, the interface returns NULLNAME.

Mach-Reply-Portp $\equiv$ Mach-Port-Allocate-Receive $\vee \uparrow(\mathbf{n} = \text{NULLNAME})[\alpha]$

Chapter 4

Virtual Memory Interface

4.1 vm_allocate

DESCRIPTION

Allocate a region of virtual memory.

PARAMETERS

- **t**. The target task.
- $\mathbf{va_1}$. A virtual address in **t**'s address space. The kernel truncates it to a page boundary.
- **l**. The length of the region of interest. The kernel rounds up to an even number of pages[1]. However, as a special case, $\mathbf{l} = \mathtt{0}$ returns immediately with a successful return code.
- **anywhere**. Placement indicator. If false, the kernel allocates the region at the virtual page address containing $\mathbf{va_1}$. If true, the kernel chooses a region wherever space is available.
- $\mathbf{va_2}$. [out] The page-aligned virtual address of a region of **t**'s address space where the memory was allocated. Pages containing the region from $\mathbf{va_2}$ to $\mathbf{va_2} + \mathbf{l}$ are affected.

OUTCOMES

The possible outcomes are *success*, *invalid task*, *invalid address*, and *no space*.

$$\begin{array}{ll} & \text{Vm-Allocatep} \\ \equiv & \quad \text{Vm-Allocate-Success} \\ & \lor\ \text{Vm-Allocate-Invalid-Argument} \\ & \lor\ \text{Vm-Allocate-Invalid-Address} \\ & \lor\ \text{Vm-Allocate-No-Space} \end{array}$$

[1]Consider a case where a task allocates ten words for a virtual address one below a page boundary. Because of the rounding down of $\mathbf{va_1}$ and the rounding up of **l**, one page will be allocated.

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task. A temporary memory m is allocated[2] and mapped into the task's address space at virtual page address $\mathbf{va_2}$. If **anywhere**, the kernel finds space wherever it is available[3], otherwise it is at $\mathbf{va_1}$'s virtual page address. The contents of the memory for the relevant region are zero-filled. Default values are provided for inheritance and protection values.

$$
\begin{array}{lllll}
 & \text{Vm-Allocate-Success} & & & \\
\equiv & \Diamond \text{taskp}\,(\mathbf{t})[\alpha] & & & \\
; & \quad \Diamond(\mathbf{l} = \mathtt{0})[\alpha] & & & \\
 & \vee\ \Diamond(& (\mathbf{l} > \mathtt{0})[\alpha] & & \\
 & & \wedge\ \text{page-aligned}\,(\mathbf{va_2})[\alpha] & & \\
 & & \wedge\ (& (\neg\ \mathbf{anywhere})[\alpha] & \\
 & & & \rightarrow (\mathbf{va_2} = \text{trunc-page}\,(\mathbf{va_1}))[\alpha]) & \\
 & & \wedge\ \text{Vm-Allocate-Zero-Mapped-Memoryp}) & & \\
; & \Diamond\uparrow(\mathbf{rc} = \mathtt{'kern\text{-}success})[\alpha] & & &
\end{array}
$$

[2]We are ignoring *coalescing* of memory entities. The implementation optimizes by coalescing the new memory entity with an adjoining one, if certain criteria are met. This optimization is visible to the user task via the `vm_region` kernel service. The implementation chooses to coalesce if

- the previous virtual page address (vpa) is allocated, and
- the memory entity at the previous vpa is temporary, and
- the memory entity at the previous vpa has exactly that one map, and
- the memory entity is not managed. (If the default pager is managing this memory entity, it is too late to change its size.)

[3]This is not consistant with vm_map, which guarentees that the allocated region is after the input address $\mathbf{va_1}$.

Vm-Allocate-Zero-Mapped-Memoryp

$$\begin{aligned}
\equiv\quad & \forall\, 0 \le i < \mathbf{l}:\\
& \quad (\Diamond(\neg\ \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(\mathbf{va_2} + i)))[\alpha])\\
;\ & \exists\, m \in \text{ALL-ENTITIES}:\\
& \quad (\quad \Diamond{\uparrow}\text{memoryp}\,(m)[\alpha]\\
& \quad ;\quad \Diamond{\uparrow}\text{temporary-rel}\,(m)[\alpha]\\
& \qquad \wedge\ \Diamond\forall\, 0 \le i < \text{trunc-page}\,(\mathbf{l} + \text{PAGESIZE}):\\
& \qquad\qquad ({\uparrow}\text{m-wordp}\,(m,\ \text{trunc-page}\,(i),\ i,\ 0)[\alpha])\\
& \qquad \wedge\ \forall\, 0 \le i < \text{trunc-page}\,(\mathbf{l} + \text{PAGESIZE}):\\
& \qquad\qquad (\Diamond{\uparrow}\text{map-rel}\,(\mathbf{t},\ m,\ \mathbf{va_2} + \text{trunc-page}\,(i),\\
& \qquad\qquad\qquad \text{trunc-page}\,(i),\ \texttt{'copy},\ \{\texttt{'read},\ \texttt{'write}\},\\
& \qquad\qquad\qquad \{\texttt{'read},\ \texttt{'write},\ \texttt{'execute}\})[\alpha]))
\end{aligned}$$

An *invalid argument* outcome results when the target task is not a task. The *invalid address* outcome results when we attempt to explicitly allocate the region (not **anywhere**), but the end of the region is past the end of the address space. The *no space* outcome results when there is no room in the target task's address space for the new region.

Vm-Allocate-Invalid-Argument

$$\equiv \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]$$

Vm-Allocate-Invalid-Address

$$\begin{aligned}
\equiv\quad & \Diamond(\quad \text{taskp}\,(\mathbf{t})[\alpha]\\
& \quad \wedge\ (\neg\ \mathbf{anywhere})[\alpha]\\
& \quad \wedge\ (\mathbf{l} > 0)[\alpha]\\
& \quad \wedge\ (\qquad \text{trunc-page}\,(\mathbf{va_1})\\
& \qquad\quad + \text{trunc-page}\,(\mathbf{l} + \text{PAGESIZE})\\
& \qquad > \text{ADDRESS-SPACE-LIMIT})[\alpha])\\
;\ & \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-address})[\alpha]
\end{aligned}$$

Vm-Allocate-No-Space

$$\begin{aligned}
\equiv\quad & \quad \Diamond((\neg\ \mathbf{anywhere})[\alpha] \wedge \text{Vm-Allocate-No-Space-Herep})\\
& \vee\ \Diamond(\mathbf{anywhere}[\alpha] \wedge \text{Vm-Allocate-No-Space-Anywherep})\\
;\ & \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-no-space})[\alpha]
\end{aligned}$$

Vm-Allocate-No-Space-Herep

$$\begin{aligned}
\equiv\ & \Diamond\exists\, 0 \le i < \mathbf{l}:\\
& \quad \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(\text{trunc-page}\,(\mathbf{va_1}) + i))[\alpha]
\end{aligned}$$

$$\begin{aligned}
&\quad \text{Vm-Allocate-No-Space-Anywherep} \\
&\equiv \Diamond\neg\ \exists\ \mathtt{0} \le vpa < (\textsc{address-space-limit} - \mathbf{l}): \\
&\qquad\quad (\quad \text{page-aligned}\,(vpa)[\alpha] \\
&\qquad\quad\ \ \wedge\ \forall\ \mathtt{0} \le i < \mathbf{l}\!:\ (\neg\ \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(vpa + i)))[\alpha])
\end{aligned}$$

4.2 vm_copy

DESCRIPTION

Copy a region of a task's address space.

PARAMETERS

- **t**. The target task.
- $\mathbf{va_1}$. The source virtual address in **t**'s address space.
- **l**. The length of the region of interest.
- $\mathbf{va_2}$. The destination virtual address in **t**'s address space.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, *protection failure*, and *invalid address*.

$$\begin{array}{ll} & \text{Vm-Copyp} \\ \equiv & \quad \text{Vm-Copy-Success} \\ & \lor\ \text{Vm-Copy-Invalid-Argument} \\ & \lor\ \text{Vm-Copy-Invalid-Address} \\ & \lor\ \text{Vm-Copy-Protection-Failure} \end{array}$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task, the source region can be read, and the destination region can be written. The kernel takes a snapshot of the contents of the source region in a state and replaces the contents of the destination region with it. The regions may overlap, and no map relations are changed. The source address, destination address, or length need not be page-aligned.

```
    Vm-Copy-Success
≡     $\forall\, 0 \le i < \mathbf{l}$:
         ($\exists$ ($0 \le w <$ WORDSIZE):
            (    $\Diamond$(    (    'read
                                 $\in$ protection ($\mathbf{t}$, trunc-page ($\mathbf{va_1} + i$)))[$\alpha$]
                          $\wedge$ va-wordp ($\mathbf{t}$, $\mathbf{va_1} + i$, $w$)[$\alpha$])
              ;  $\Diamond$(    (    'write
                                 $\in$ protection ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$)))[$\alpha$]
                          $\wedge$ $\uparrow$va-wordp ($\mathbf{t}$, $\mathbf{va_2} + i$, $w$)[$\alpha$])))
     ;  $\uparrow$($\mathbf{rc}$ = 'kern-success)[$\alpha$]
```

An *invalid argument* outcome results when the target task is not a task, or when one of the arguments $\mathbf{va_1}$, $\mathbf{va_2}$, or $\mathbf{l}$ are not page-aligned[4]. The *invalid address* outcome results when there is a virtual address between $\mathbf{va_1}$ and $\mathbf{va_1} + \mathbf{l}$, or between $\mathbf{va_2}$ and $\mathbf{va_2} + \mathbf{l}$, that is not a valid address. The *protection-failure* outcome results when the source region is protected against reading or the destination region is protected against writing.

```
    Vm-Copy-Invalid-Argument
≡     $\Diamond$(    ($\neg$ taskp ($\mathbf{t}$))[$\alpha$]
         $\vee$  ($\neg$ page-aligned ($\mathbf{va_1}$))[$\alpha$]
         $\vee$  ($\neg$ page-aligned ($\mathbf{va_2}$))[$\alpha$]
         $\vee$  ($\neg$ page-aligned ($\mathbf{l}$))[$\alpha$])
     ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-argument)[$\alpha$]

    Vm-Copy-Invalid-Address
≡ $\exists\, 0 \le i < \mathbf{l}$:
      (        $\Diamond$(    taskp ($\mathbf{t}$)
                    $\wedge$ $\neg$ allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_1} + i$)))[$\alpha$]
           $\vee$  $\Diamond$(    taskp ($\mathbf{t}$)
                    $\wedge$ $\neg$ allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$)))[$\alpha$]
        ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-address)[$\alpha$])
```

[4]Formerly, the *invalid argument* outcome would also result when one of $\mathbf{va_1}$, $\mathbf{va_2}$, or $\mathbf{l}$ was not page-aligned. As of August, 1993, this restriction has been partially removed from the implementation. Our statement of the specification allows an implementation to either succeed or fail in this case.

$$
\begin{array}{l}
\quad \text{Vm-Copy-Protection-Failure} \\
\equiv \exists\ \mathtt{0} \le i < \mathbf{l}\text{:} \\
\quad (\qquad \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va_1} + i))[\alpha] \\
\qquad\qquad\qquad \wedge\ (\quad \mathtt{'read} \\
\qquad\qquad\qquad\qquad \notin\ \text{protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va_1} + i)))[\alpha]) \\
\qquad\quad \vee\ \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va_2} + i))[\alpha] \\
\qquad\qquad\qquad \wedge\ (\quad \mathtt{'write} \\
\qquad\qquad\qquad\qquad \notin\ \text{protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va_2} + i)))[\alpha]) \\
\qquad ;\ \Diamond{\uparrow}(\mathbf{rc} = \mathtt{'kern\text{-}protection\text{-}failure})[\alpha])
\end{array}
$$

4.3 vm_deallocate

DESCRIPTION

Deallocate a region of virtual memory.

PARAMETERS

- **t**. The target task.
- **va**. A virtual address in **t**'s address space.
- **l**. The length of the region of interest. All pages which contain data in the region **va** to **va** + **l** are affected.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, and *invalid address*.

$$\begin{array}{ll} & \text{Vm-Deallocatep} \\ \equiv & \quad \text{Vm-Deallocate-Success} \\ & \vee\ \text{Vm-Deallocate-Invalid-Argument} \\ & \vee\ \text{Vm-Deallocate-Invalid-Address} \end{array}$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and all addresses in the region are removed. When the last map for a memory entity is deallocated, the memory entity is killed.

```
    Vm-Deallocate-Success
≡       $\Diamond$taskp ($\mathbf{t}$)[$\alpha$]
    ;  $\forall$ 0 $\leq i <$ $\mathbf{l}$:
         ($\Diamond$(     allocated ($\mathbf{t}$, $\mathbf{va} + i$)[$\alpha$]
              $\rightarrow$      $\Diamond\downarrow$allocated ($\mathbf{t}$, $\mathbf{va} + i$)[$\alpha$]
                   $\wedge$ $\exists\, m$ = mapped-memory ($\mathbf{t}$, trunc-page ($\mathbf{va} + i$)):
                        (     (mapping-tasks ($m$) = $\{\mathbf{t}\}$)[$\alpha$]
                         $\rightarrow$ Terminate-Memory ($m$))))
    ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-success)[$\alpha$]
```

An *invalid argument* outcome results when the target task is not a task. The *invalid address* outcome results when there is a virtual address between **va** and **va** +**l** that is not a valid address[5].

```
    Vm-Deallocate-Invalid-Argument
≡ $\Diamond$($\neg$ taskp ($\mathbf{t}$))[$\alpha$] ; $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-argument)[$\alpha$]

    Vm-Deallocate-Invalid-Address
≡ $\exists$ 0 $\leq i <$ $\mathbf{l}$:
     (     $\Diamond$(taskp ($\mathbf{t}$) $\wedge\neg$ allocated ($\mathbf{t}$, trunc-page ($\mathbf{va} + i$)))[$\alpha$]
       ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-address)[$\alpha$])
```

[5]Contrary to [Loe91], the implementation never returns `'kern-invalid-address`. It just skips holes in the region. There was a series of email messages about this, starting with "VM Kernel Interface Semantics" from dlb, 27 Jan 92. Rashid and others stated the strong opinion that holes should be allowed. Our specification allows the implementation to either ignore the holes or fail.

4.4 vm_inherit

DESCRIPTION

Set the inheritance attribute for a region of virtual memory.

PARAMETERS

- **t**. The target task.
- **va**. A virtual address in **t**'s address space.
- **l**. The length of the region of interest.
- **inh**. The new inheritance value, one of {`'share`, `'copy`, `'none`}.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, and *invalid address.*

$$
\begin{array}{ll}
 & \text{Vm-Inheritp} \\
\equiv & \quad \text{Vm-Inherit-Success} \\
 & \vee\ \text{Vm-Inherit-Invalid-Argument} \\
 & \vee\ \text{Vm-Inherit-Invalid-Address}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the allocated virtual page addresses in the region are given the specified inheritance.

$$
\begin{array}{l}
\quad \text{Vm-Inherit-Success} \\
\equiv \forall\ \mathtt{0} \le i < \mathbf{l}: \\
\quad\quad (\quad \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i))[\alpha] \\
\quad\quad\quad\quad\quad \rightarrow \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)) \\
\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ (\text{inheritance}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)) = \mathbf{inh}))[\alpha]) \\
\quad\quad\quad ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha])
\end{array}
$$

An *invalid argument* outcome results when either the target task is not a task, or the new inheritance value is not valid. The *invalid address* outcome results when there is a virtual address between **va** and **va** +**l** that is not allocated[6].

$$
\begin{array}{ll}
 & \text{Vm-Inherit-Invalid-Argument} \\
\equiv & \quad \Diamond(\neg\, \text{taskp}\,(\mathbf{t}))[\alpha] \vee \Diamond(\mathbf{inh} \notin \mathcal{I})[\alpha] \\
 & ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Vm-Inherit-Invalid-Address} \\
\equiv \exists\, \texttt{0} \leq i < \mathbf{l}\text{:} \\
\quad (\quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\, \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)))[\alpha] \\
\quad\quad ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-address})[\alpha])
\end{array}
$$

[6] The documentation ([Loe91]) says that *invalid address* is returned if there is a non-allocated region in the range from **va** to **va** +**l**. The implementation ignores holes in the region. Our specification allows both behaviors.

4.5 vm_map

DESCRIPTION

Map a memory object into a task's address space.

PARAMETERS

- **t**. The target task.
- $\mathbf{va_1}$. A virtual address in **t**'s address space which is the start of the region of interest. The kernel truncates to a page boundary.
- **l**. The length of the region of interest. Enough pages are allocated to cover **l** bytes.
- **anywhere**. Placement indicator. If false, the kernel allocates the region at the virtual page address containing **va**. If true, the kernel chooses a region wherever space is available[7].
- **n**. A local port name representing the memory entity of interest.
- **offset**. The offset into the memory entity to which we wish to map the address region.
- **copy**. A boolean flag that indicates that we wish to take a snapshot of the memory entity rather than mapping it directly.
- **CP**. Current protection set for the new region.
- **MP**. Maximum protection set for the new region.
- **inh**. Inheritance value for the new region.
- $\mathbf{va_2}$. [out] The page-aligned virtual address where the map is actually assigned.

[7]The implementation provides an additional argument **mask**, which provides alignment restrictions for the starting address.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, and *no space*.

$$\text{Vm-Mapp} \equiv \text{Vm-Map-Success} \vee \text{Vm-Map-Invalid-Argument} \vee \text{Vm-Map-No-Space}$$

SPECIFICATION

On a successful outcome, the virtual page address in the target task is chosen, the memory entity and offset of interest is chosen, then the range of the memory entity is mapped into the task's address space.

$$\begin{array}{l}
\quad \text{Vm-Map-Success} \\
\equiv \exists\, m \in \textsc{all-entities},\ 0 \leq o < \textsc{memorysize}: \\
\qquad (\quad \Diamond\text{Vm-Map-Va2-Selectp} \\
\qquad\ \ \wedge\ \Diamond\text{Vm-Map-Memory-Selectp}\,(m,\ o) \\
\qquad\ \ \wedge\ \forall\, 0 \leq i < \mathbf{l}: \\
\qquad\qquad (\Diamond{\uparrow}\text{map-rel}\,(\mathbf{t},\ m,\ \mathbf{va_2} + \text{trunc-page}\,(i), \\
\qquad\qquad\qquad\qquad o + \text{trunc-page}\,(i),\ \mathbf{inh},\ \mathbf{CP},\ \mathbf{MP})[\alpha]) \\
\qquad\ \ \wedge\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha])
\end{array}$$

The mapped virtual page address in the target task is chosen according to the input parameters $\mathbf{t}$, $\mathbf{va_1}$, $\mathbf{l}$, and **anywhere**[8].

$$\begin{array}{l}
\quad \text{Vm-Map-Va2-Selectp} \\
\equiv \qquad \text{page-aligned}\,(\mathbf{va_2})[\alpha] \\
\quad\ \wedge\ (\text{trunc-page}\,(\mathbf{va_1}) \leq \mathbf{va_2})[\alpha] \\
\quad\ \wedge\ (\quad (\neg\ \mathbf{anywhere})[\alpha] \\
\qquad\quad \rightarrow (\mathbf{va_2} = \text{trunc-page}\,(\mathbf{va_1}))[\alpha]) \\
\quad\ \wedge\ \forall\ \mathbf{va_2} \leq va < (\mathbf{va_2} + \mathbf{l}): \\
\qquad\quad (\Diamond(\neg\ \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(va)))[\alpha])
\end{array}$$

The algorithm for determining the memory entity of interest from the local port name $\mathbf{n}$ has a number of cases.

[8]The treatment of **anywhere** is not consistent between `vm_allocate` and `vm_map`. For `vm_map`, the allocated space is guaranteed to be at or after the input value $\mathbf{va_1}$.

- If **n** $\neq$NULLNAME and **copy** is false, then **n** names a send right to a port. If the port is an object port for a memory entity, then it is the one of interest. Otherwise, a new memory entity is created and the kernel enters into an initialization dialog with the task which holds the receive right for the port. That task is the *external memory manager* for the memory entity. We do not specify the details of that dialog here.

- If **n** =NULLNAME, then m is a new, temporary memory which is zero-filled.

- If **copy** is true, then **n** names a port as for the first case. A snapshot of the region of interest for that port's object memory is copied into a new, temporary memory.

$$
\begin{array}{ll}
 & \text{Vm-Map-Memory-Selectp}\,(m,\ o) \\
\equiv & \quad \text{Vm-Map-Memory-Select-S-Rightp}\,(m,\ o) \\
 & \vee\ \text{Vm-Map-Memory-Select-Nullnamep}\,(m,\ o) \\
 & \vee\ \text{Vm-Map-Memory-Select-Copyp}\,(m,\ o)
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Vm-Map-Memory-Select-S-Rightp}\,(m,\ o) \\
\equiv & \exists\, p \in \text{ALL-ENTITIES:} \\
 & \quad (\quad (\neg\ \mathbf{copy})[\alpha] \\
 & \quad\ \ \wedge\ \text{s-right}\,(\mathbf{t},\ \mathbf{n})[\alpha] \\
 & \quad\ \ \wedge\ (p = \text{named-port}\,(\mathbf{t},\ \mathbf{n}))[\alpha] \\
 & \quad\ \ \wedge\ \Diamond\text{object-port-rel}\,(m,\ p)[\alpha] \\
 & \quad\ \ \wedge\ (o = \mathbf{offset})[\alpha])
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Vm-Map-Memory-Select-Nullnamep}\,(m,\ o) \\
\equiv & \quad (\mathbf{n} = \text{NULLNAME})[\alpha] \\
 & \wedge\ \Diamond\!\uparrow\!\text{memoryp}\,(m)[\alpha] \\
 & \wedge\ \Diamond\!\uparrow\!\text{temporary-rel}\,(m)[\alpha] \\
 & \wedge\ (o = 0)[\alpha]
\end{array}
$$

```
   Vm-Map-Memory-Select-Copyp $(m,\ o)$
$\equiv \exists\ p \in$ ALL-ENTITIES, $m_1 \in$ ALL-ENTITIES:
    (     $\mathbf{copy}[\alpha]$
      $\wedge$ s-right $(\mathbf{t},\ \mathbf{n})[\alpha]$
      $\wedge$ $(p =$ named-port $(\mathbf{t},\ \mathbf{n}))[\alpha]$
      $\wedge$ $\Diamond$object-port-rel $(m_1,\ p)[\alpha]$
      $\wedge$ $\Diamond\uparrow$memoryp $(m)[\alpha]$
      $\wedge$ $\Diamond\uparrow$temporary-rel $(m)[\alpha]$
      $\wedge$ $(o = \mathtt{0})[\alpha]$
      $\wedge$ $\forall\ \mathtt{0} \leq i <$ trunc-page $(\mathbf{l} + \text{PAGESIZE})$:
           $(\exists\ (\mathtt{0} \leq w < \text{WORDSIZE})$:
               (     m-wordp $(m_1,\ \mathbf{offset} +$ trunc-page $(i),\ \mathbf{offset} + i,$
                                 $w)[\alpha]$
                 $\wedge$ $\Diamond$m-wordp $(m,$ trunc-page $(i),$ 'i, 'w$)[\alpha])))$
```

An *invalid argument* outcome results in a number of cases. Either

- the target task is not a task,
- The current or maximum protections are not in the proper set,
- the current protection is not a subset of the maximum protection,
- the inheritance value is not in the proper set, or
- the local name is a name for a port which is a special port, but not a memory object port[9].

The *no space* outcome results when there is no room in the target task's address space for the new region.

[9]The predicate t(:fnname port-is-special-port) recognizes when its argument is any of the possible kinds of special ports.

$$
\begin{array}{ll}
& \text{Vm-Map-Invalid-Argument} \\
\equiv & \qquad \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\
& \quad \lor\ \Diamond(\mathbf{CP} \notin \mathcal{P})[\alpha] \\
& \quad \lor\ \Diamond(\mathbf{CP} \notin \mathcal{P})[\alpha] \\
& \quad \lor\ \Diamond(\mathbf{CP} \not\subseteq \mathbf{MP})[\alpha] \\
& \quad \lor\ \Diamond(\mathbf{inh} \notin \mathcal{I})[\alpha] \\
& \quad \lor\ (\quad \Diamond\text{port-right-namep}\,(\mathbf{t},\, \mathbf{n})[\alpha] \\
& \qquad\quad ;\quad \text{Port-Is-Special-Port}\,(\text{named-port}\,(\mathbf{t},\, \mathbf{n})) \\
& \qquad\qquad \land\ \neg\ \text{exists-control-memory}\,(\text{named-port}\,(\mathbf{t},\, \mathbf{n}))[\alpha]) \\
& ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Map-No-Space} \\
\equiv & \quad \Diamond(\quad (\neg\ \mathbf{anywhere})[\alpha] \land \text{Vm-Map-No-Space-Herep} \\
& \qquad \lor\ \mathbf{anywhere}[\alpha] \land \text{Vm-Map-No-Space-Anywherep}) \\
& ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-no-space})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Map-No-Space-Herep} \\
\equiv & \Diamond\exists\ \mathtt{0} \le i < \mathbf{l}\text{:} \\
& \quad \text{allocated}\,(\mathbf{t},\, \text{trunc-page}\,(\text{trunc-page}\,(\mathbf{va_1}) + i))[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Map-No-Space-Anywherep} \\
\equiv & \Diamond\neg\ \exists\ \mathtt{0} \le \mathit{vpa} < (\textsc{address-space-limit} - \mathbf{l})\text{:} \\
& \qquad (\quad \text{page-aligned}\,(\mathit{vpa})[\alpha] \\
& \qquad\ \land\ \forall\ \mathtt{0} \le i < \mathbf{l}\text{:}\ (\neg\ \text{allocated}\,(\mathbf{t},\, \text{trunc-page}\,(\mathit{vpa} + i)))[\alpha])
\end{array}
$$

4.6 vm_protect

DESCRIPTION

Change the protection or maximum protection for a region of a task's address space.

PARAMETERS

- **t**. The target task.
- **va**. A virtual address in **t**'s address space which is the start of the region of interest.
- **l**. The length of the region of interest. All pages which contain data in the region from **va** to **va** + **l** are affected.
- **set-maximum**. If true, the maximum protection is set. If false, the current protection is set.
- **inh**. The new inheritance value, a subset of {`'read`, `'write`, `'execute`}.

OUTCOMES

The possible outcomes are *success*, *protection failure*, *invalid argument*, and *invalid address*.

$$
\begin{array}{ll}
 & \text{Vm-Protectp} \\
\equiv & \quad \text{Vm-Protect-Success} \\
 & \lor\ \text{Vm-Protect-Invalid-Argument} \\
 & \lor\ \text{Vm-Protect-Invalid-Address} \\
 & \lor\ \text{Vm-Protect-Protection-Failure}
\end{array}
$$

SPECIFICATION

On a successful outcome, the target task is confirmed to be a task and the maximum or current protection is set.

$$
\begin{array}{l}
\quad \text{Vm-Protect-Success} \\
\equiv \quad \Diamond(\quad \text{taskp}\,(\mathbf{t})[\alpha] \\
\qquad\qquad \wedge\ (\textbf{set-maximum}[\alpha] \rightarrow \text{Vm-Protect-Maximump}) \\
\qquad\qquad \wedge\ ((\neg\ \textbf{set-maximum})[\alpha] \rightarrow \text{Vm-Protect-Currentp})) \\
\quad\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Vm-Protect-Maximump} \\
\equiv \forall\ 0 \leq i < \mathbf{l}: \\
\quad (\Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i))[\alpha] \\
\qquad \rightarrow \exists\ CP = \text{protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)), \\
\qquad\qquad\quad MP = \text{max-protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)): \\
\qquad\qquad (\Diamond(\quad (\mathbf{PR} \subseteq MP)[\alpha] \\
\qquad\qquad\quad \wedge\ \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i))[\alpha] \\
\qquad\qquad\quad \wedge\ (\text{max-protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)) = \mathbf{PR})[\alpha] \\
\qquad\qquad\quad \wedge\ (\quad (CP \not\subseteq \mathbf{PR})[\alpha] \\
\qquad\qquad\qquad \rightarrow \uparrow(\text{protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)) = \mathbf{PR})[\alpha])))))
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Vm-Protect-Currentp} \\
\equiv \forall\ 0 \leq i < \mathbf{l}: \\
\quad (\Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i))[\alpha] \\
\qquad \rightarrow \quad (\mathbf{PR} \subseteq \text{max-protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)))[\alpha] \\
\qquad\quad \wedge\ \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i))[\alpha] \\
\qquad\qquad \wedge\ (\text{protection}\,(\mathbf{t}, \text{trunc-page}\,(\mathbf{va} + i)) = \mathbf{PR})[\alpha])))
\end{array}
$$

An *invalid argument* outcome results when the target task is not a task. The *invalid address* outcome results when there is a virtual address between **va** and **va** +**l** that is not a valid address[10]. The *protection-failure* outcome results when the new protection is greater than a region's maximum protection.

$$
\begin{array}{l}
\quad \text{Vm-Protect-Invalid-Argument} \\
\equiv \quad \Diamond(\quad \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\
\qquad\qquad \vee\ (\mathbf{PR} \subseteq \{\texttt{'read}, \texttt{'write}, \texttt{'execute}\})[\alpha]) \\
\quad\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}
$$

[10]The documentation ([Loe91]) says that *invalid address* is returned if there is a non-allocated region in the range from **va** to **va** +**l**. The implementation ignores holes in the region. Our specification allows both behaviors.

$$
\begin{aligned}
&\quad \text{Vm-Protect-Invalid-Address} \\
\equiv\ & \exists\, \mathtt{0} \le i < \mathbf{l}: \\
&\quad (\quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\ \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(\mathbf{va} + i)))[\alpha] \\
&\quad\ \ ;\ \ \Diamond\!\uparrow\!(\mathbf{rc} = \mathtt{'kern\text{-}invalid\text{-}address})[\alpha])
\end{aligned}
$$

$$
\begin{aligned}
&\quad \text{Vm-Protect-Protection-Failure} \\
\equiv\ & \exists\, \mathtt{0} \le i < \mathbf{l}: \\
&\quad (\quad \Diamond(\quad \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(\mathbf{va} + i)) \\
&\qquad\qquad \wedge\ \mathbf{PR} \not\subseteq \text{max-protection}\,(\mathbf{t},\ \text{trunc-page}\,(\mathbf{va} + i)))[\alpha] \\
&\quad\ \ ;\ \ \Diamond\!\uparrow\!(\mathbf{rc} = \mathtt{'kern\text{-}protection\text{-}failure})[\alpha])
\end{aligned}
$$

4.7 vm_read

DESCRIPTION

Read a task's virtual memory. The effect of vm_read is as if the target task has mailed the memory segment to the agent task.

PARAMETERS

- **ct**. The agent (current) task. In the implementation, this argument is implicit.
- **t**. The target task.
- **va**. A virtual address in **t**'s address space which is the start of the region of interest.
- **l**. The length of the region of interest. The region read contains all pages in the range **va** to $\mathbf{va_1} + \mathbf{l}$.
- $\mathbf{va_2}$. [out] The virtual page address in **ct** where the data is placed. The length if the region is **l**, rounded up to a page boundary. This length value is an explicit **OUT** parameter in the implementation.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, *invalid address*, *protection-failure*, and *no-space*.

$$
\begin{array}{ll}
 & \text{Vm-Readp} \\
\equiv & \quad \text{Vm-Read-Success} \\
 & \lor\ \text{Vm-Read-Invalid-Argument} \\
 & \lor\ \text{Vm-Read-Invalid-Address} \\
 & \lor\ \text{Vm-Read-Protection-Failure} \\
 & \lor\ \text{Vm-Read-No-Space}
\end{array}
$$

SPECIFICATION

On a successful outcome, the address space in the target task **t** is found to be allocated, with reading allowed. The data in the region is copied to a newly-allocated area in **ct**.

```
    Vm-Read-Success
 ≡ ∀ 0 ≤ i < l:
      (∃ (0 ≤ w < WORDSIZE):
         (◇(    allocated (t, trunc-page (va₁ + i))[α]
             ∧  ('read ∈ protection (t, trunc-page (va₁ + i)))[α]
             ∧  va-wordp (t, va₁ + i, w)[α]
             ∧  page-aligned (va₂)[α]
             ∧  ◇↑allocated (ct,
                              trunc-page (    va₂
                                          + (va₁ − trunc-page (va₁))
                                          + i))[α]
             ∧  ◇↑va-wordp (ct,
                                 va₂
                               + (va₁ − trunc-page (va₁))
                               + i, w)[α]
             ∧  ◇↑(rc = 'kern-success)[α])))
```

An *invalid argument* outcome results when the target task is not a task, or if the input **va** or **l** are not page-aligned[11]. The *invalid address* outcome results when there is a virtual address between **va** and **va** +**l** that is not allocated. The *protection failure* outcome results when a part of **t**'s region is protected against reading. The *no space* outcome results when there is not enough room in **ct**'s address space to place the data.

```
    Vm-Read-Invalid-Argument
 ≡ ◇(¬ taskp (t))[α] ; ◇↑(rc = 'kern-invalid-argument)[α]
```

[11]The documentation ([Loe91]) states that the arguments **va** and **l** must be page-aligned. The implementation does not enforce this restriction. Our specification allows both behaviors.

$$
\begin{array}{l}
\quad \text{Vm-Read-Invalid-Address} \\
\equiv \exists\, \mathbf{va_1} \le va < (\mathbf{va_1} + \mathbf{l}): \\
\quad (\quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge \neg\, \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(va)))[\alpha] \\
\quad\quad ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-address})[\alpha])
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Vm-Read-Protection-Failure} \\
\equiv \exists\, \mathbf{va_1} \le va < (\mathbf{va_1} + \mathbf{l}): \\
\quad (\quad \Diamond(\quad \text{allocated}\,(\mathbf{t}, \text{trunc-page}\,(va)) \\
\qquad\qquad \wedge\ \texttt{'read} \notin \text{protection}\,(\mathbf{t}, \text{trunc-page}\,(va)))[\alpha] \\
\quad\quad ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-protection-failure})[\alpha])
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Vm-Read-No-Space} \\
\equiv \quad \Diamond\neg\, \exists\, \mathtt{0} \le va < \text{ADDRESS-SPACE-LIMIT}: \\
\qquad\qquad (\forall\, (\mathtt{0} \le i < \mathbf{l}): \\
\qquad\qquad\quad (\neg\, \text{allocated}\,(\mathbf{ct}, \text{trunc-page}\,(va + i)))[\alpha]) \\
\quad ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-no-space})[\alpha]
\end{array}
$$

4.8 vm_region

DESCRIPTION

Return information about a region of a task's virtual memory.

PARAMETERS

- **t**. The target task.
- **va**. [in] A virtual address in **t**'s address space to begin looking for a region.
- **vpa**. [out] The beginning of the region found.
- **l**. [out] The length of the region of interest. The kernel rounds up to a page boundary.
- **CP**. [out] Current protection for the region.
- **MP**. [out] Maximum protection for the region.
- **inh**. [out] Inheritance value for the region
- **shared**. [out] Boolean flag, set if the region is shared by another task.
- **n**. [out] The task's local name for the name port of the memory entity associated with this region, if any.
- **o**. [out] The offset into the memory entity which is associated with the beginning of this region.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, and *no-space*.

$$\begin{aligned}&\text{Vm-Regionp}\\ \equiv\ &\text{Vm-Region-Success} \lor \text{Vm-Region-Invalid-Argument} \lor \text{Vm-Region-No-Space}\end{aligned}$$

SPECIFICATION

On a successful outcome, the returned argument **vpa** is the virtual page address of the allocated region containing or following **va**. All virtual page addresses from **vpa** to **vpa** + **l** have the same characteristics[12]. The **shared** flag tells whether some other child or sibling task has inherited the region.

$$
\begin{array}{ll}
& \text{Vm-Region-Success} \\
\equiv & \quad \Diamond\text{Vm-Region-Within} \vee \Diamond\text{Vm-Region-Next} \\
& ;\ \text{Vm-Region-Outputs} \\
& ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'success})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Region-Within} \\
\equiv & \quad \text{allocated}\,(\mathbf{t},\ \mathbf{va})[\alpha] \\
& \wedge\ \text{page-aligned}\,(\mathbf{vpa})[\alpha] \\
& \wedge\ (\mathbf{vpa} \leq \mathbf{va} < (\mathbf{vpa} + \mathbf{l}))[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Region-Next} \\
\equiv & \quad (\neg\ \text{allocated}\,(\mathbf{t},\ \mathbf{va}))[\alpha] \\
& \wedge\ \text{page-aligned}\,(\mathbf{vpa})[\alpha] \\
& \wedge\ (\mathbf{va} < \mathbf{vpa})[\alpha] \\
& \wedge\ \forall\ \mathbf{va} \leq \mathit{va1} < \mathbf{vpa}\text{:}\ \ (\neg\ \text{allocated}\,(\mathbf{t},\ \mathit{va1}))[\alpha] \\
& \wedge\ \text{allocated}\,(\mathbf{t},\ \mathbf{vpa})[\alpha]
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Region-Outputs} \\
\equiv & \exists\ m \in \text{ALL-ENTITIES:} \\
& \quad (\quad \text{map-rel}\,(\mathbf{t},\ m,\ \mathbf{vpa},\ \mathbf{o},\ \mathbf{inh},\ \mathbf{CP},\ \mathbf{MP})[\alpha] \\
& \quad\ \ \wedge\ \text{Vm-Region-Range}\,(m) \\
& \quad\ \ \wedge\ \text{Vm-Region-Name-Port-Selected}\,(m) \\
& \quad\ \ \wedge\ \text{Vm-Region-Shared-Flag-Set}\,(m))
\end{array}
$$

$$
\begin{array}{ll}
& \text{Vm-Region-Range}\,(m) \\
\equiv & \exists\ 0 \leq i < \mathbf{l}\text{:} \\
& \quad (\quad \text{page-aligned}\,(i)[\alpha] \\
& \quad\ \ \rightarrow \text{map-rel}\,(\mathbf{t},\ m,\ \mathbf{vpa} + i,\ \mathbf{o} + i,\ \mathbf{inh},\ \mathbf{CP},\ \mathbf{MP})[\alpha])
\end{array}
$$

[12]In the implementation, it may be that the following virtual page address has the identical attributes. That is, the given region is not the largest posible with those attributes. The implementation defines a region with the `vm_entry` data structure.

$$\begin{array}{ll} & \text{Vm-Region-Name-Port-Selected}\,(m) \\ \equiv & \text{exists-name-port}\,(m)[\alpha] \rightarrow \text{Insert-Send-Right}\,(\mathbf{t},\ \text{name-port}\,(m),\ \mathbf{n}) \end{array}$$

$$\begin{array}{ll} & \text{Vm-Region-Shared-Flag-Set}\,(m) \\ \equiv & \quad \exists\, t_1 \in \textsc{all-entities},\ 0 \leq \mathit{vpa1} < \textsc{address-space-limit}: \\ & \qquad (\quad (t_1 \neq \mathbf{t})[\alpha] \\ & \qquad\ \ \wedge\ \text{allocated}\,(t_1,\ \mathit{vpa1})[\alpha] \\ & \qquad\ \ \wedge\ (\text{mapped-memory}\,(t_1,\ \text{trunc-page}\,(\mathit{vpa1})) = m)[\alpha] \\ & \qquad\ \ \wedge\ (\text{mapped-offset}\,(t_1,\ \text{trunc-page}\,(\mathit{vpa1})) = \mathbf{o})[\alpha]) \\ & \rightarrow \Diamond{\uparrow}\mathbf{shared}[\alpha] \end{array}$$

An *invalid argument* outcome results when the target task is not a task. The *no-space* outcome results when there are no more allocated regions following **va**.

$$\begin{array}{ll} & \text{Vm-Region-Invalid-Argument} \\ \equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha] \end{array}$$

$$\begin{array}{ll} & \text{Vm-Region-No-Space} \\ \equiv & \quad \Diamond\neg\ \exists\ \mathbf{va} \leq \mathit{va1} < \textsc{address-space-limit}: \\ & \qquad\quad \text{allocated}\,(\mathbf{t},\ \text{trunc-page}\,(\mathit{va1}))[\alpha] \\ & \ ;\ \ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-no-space})[\alpha] \end{array}$$

4.9 vm_write

SPECIFICATION

Copy a region of data from the agent (current) task to the target task.

PARAMETERS

- **ct**. The agent (current) task.
- $\mathbf{va_1}$. An address in **ct**'s address space.
- **t**. The target task.
- $\mathbf{va_2}$. A virtual address in **t**'s address space.
- **l**. The length of the region of interest.

OUTCOMES

The possible outcomes are *success*, *invalid argument*, *invalid address*, and *protection-failure*.

$$\begin{array}{ll} & \text{Vm-Writep} \\ \equiv & \quad \text{Vm-Write-Success} \\ & \lor\ \text{Vm-Write-Invalid-Argument} \\ & \lor\ \text{Vm-Write-Invalid-Address} \\ & \lor\ \text{Vm-Write-Protection-Failure} \end{array}$$

SPECIFICATION

On a successful outcome, the source region in **ct** is copied to the destination region in **t**.

```
    Vm-Write-Success
≡     $\forall\ 0 \leq i < \mathbf{l}$:
        ($\exists$ ($0 \leq w <$ WORDSIZE):
          (    $\Diamond$(    allocated ($\mathbf{ct}$, trunc-page ($\mathbf{va_1} + i$))[$\alpha$]
                  $\wedge$ (    'read
                            $\in$ protection ($\mathbf{ct}$, trunc-page ($\mathbf{va_1} + i$)))[$\alpha$]
                  $\wedge$ va-wordp ($\mathbf{ct}$, $\mathbf{va_1} + i$, $w$)[$\alpha$])
             ;  allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$))[$\alpha$]
             ;  $\uparrow$va-wordp ($\mathbf{t}$, $\mathbf{va_2} + i$, $w$)[$\alpha$]))
     ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-success)[$\alpha$]
```

An *invalid argument* outcome results when the target task is not a task. The *invalid address* outcome results when there is a virtual address between **va** and **va** + **l** that is not a valid address. The *protection-failure* outcome occurs when either the source region cannot be read or the destination region cannot be written.

```
    Vm-Write-Invalid-Argument
≡ $\Diamond$($\neg$ taskp ($\mathbf{t}$))[$\alpha$] ; $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-argument)[$\alpha$]

    Vm-Write-Invalid-Address
≡ $\exists\ 0 \leq i < \mathbf{l}$:
    (        $\Diamond$(    taskp ($\mathbf{ct}$)
                $\wedge$ $\neg$ allocated ($\mathbf{ct}$, trunc-page ($\mathbf{va_1} + i$)))[$\alpha$]
         $\vee$  $\Diamond$(    taskp ($\mathbf{t}$)
                $\wedge$ $\neg$ allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$)))[$\alpha$]
      ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-invalid-address)[$\alpha$])

    Vm-Write-Protection-Failure
≡ $\exists\ 0 \leq i < \mathbf{l}$:
    (        $\Diamond$(    allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_1} + i$))
                $\wedge$     'read
                      $\notin$ protection ($\mathbf{t}$, trunc-page ($\mathbf{va_1} + i$)))[$\alpha$]
         $\vee$  $\Diamond$(    allocated ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$))
                $\wedge$     'write
                      $\notin$ protection ($\mathbf{t}$, trunc-page ($\mathbf{va_2} + i$)))[$\alpha$]
      ;  $\Diamond\uparrow$($\mathbf{rc}$ = 'kern-protection-failure)[$\alpha$])
```

Chapter 5

Thread Interface

5.1 mach_thread_self

DESCRIPTION

Look up the sself port for a thread. Kernel services on behalf of a thread are requested by sending messages to its self port. The sself port is usually the same as the self port, unless a debugging task has interposed itself.

PARAMETERS

th. The thread of interest.

p. [out] The thread's sself port.

OUTCOMES

The implementation operates implicitly on the currently executing thread. The return code is also implicit; it expects the outcome to always be *success*.

SPECIFICATION

On a successful outcome (the only kind), input argument **th** is found to be a thread, and then its sself port is found. If the thread happens not to have a sself port, then NULL-PTR is returned.

$$\begin{array}{rl} & \text{Mach-Thread-Self-Success} \\ \equiv & \quad \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\ ; & \text{Mach-Thread-Self-Found} \\ ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

$$\begin{array}{rl} & \text{Mach-Thread-Self-Found} \\ \equiv & \quad ((\neg\ \text{exists-thread-sself}\,(\mathbf{th}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{p} = \text{NULL-PTR})[\alpha]) \\ \vee & (\text{exists-thread-sself}\,(\mathbf{th})[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{p} = \text{thread-sself}\,(\mathbf{th}))[\alpha]) \end{array}$$

5.2 thread_create

DESCRIPTION

Create and initialize a new thread, and associate it with a given target task.

PARAMETERS

t. The parent task.

th. [out] The new thread.

OUTCOMES

We specify four possible outcomes: *success*, *failure*, *invalid argument*, and *resource shortage*.

$$\begin{array}{ll} & \text{Thread-Createp} \\ \equiv & \quad \text{Thread-Create-Success} \\ & \lor\ \text{Thread-Create-Failure} \\ & \lor\ \text{Thread-Create-Invalid-Argument} \\ & \lor\ \text{Thread-Create-Resource-Shortage} \end{array}$$

SPECIFICATION

On a successful outcome, the thread is created and initialized. The self port is created, and the sself port is initially the self port. The eport is not created. It defaults to the task's eport (if any). The thread's processor set is inherited from its owning task, if it is assigned to one. If not, the thread is assigned to the default processor set.

$$\begin{array}{ll} & \text{Thread-Create-Success} \\ \equiv & \quad \Diamond\text{taskp}\,(\mathbf{t})[\alpha] \\ & ;\ \Diamond\uparrow\text{threadp}\,(\mathbf{th})[\alpha] \\ & ;\ \text{Thread-Initialized} \\ & ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

$$\begin{array}{lll}
& \multicolumn{2}{l}{\text{Thread-Initialized}} \\
\equiv & & \text{Thread-Initialized-Self-Port} \\
& \wedge & \text{Thread-Initialized-Eport} \\
& \wedge & \text{Thread-Initialized-Procset} \\
& \wedge & \Diamond\!\uparrow\!\text{task-thread-rel}\,(\mathbf{t},\ \mathbf{th})[\alpha]
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Initialized-Self-Port} \\
\equiv & \exists\ p \in \textsc{all-entities}\text{:} \\
& \quad (\quad \Diamond\!\uparrow\!\text{portp}\,(p)[\alpha] \\
& \quad\ \ ;\ \ \Diamond\!\uparrow\!\text{thread-self-rel}\,(\mathbf{th},\ p)[\alpha] \wedge \Diamond\!\uparrow\!\text{thread-sself-rel}\,(\mathbf{th},\ p)[\alpha])
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Initialized-Eport} \\
\equiv & \quad\ \exists\ p \in \textsc{all-entities}\text{:} \\
& \qquad (\Diamond\text{task-eport-rel}\,(\mathbf{t},\ p)[\alpha]\ ;\ \Diamond\!\uparrow\!\text{thread-eport-rel}\,(\mathbf{th},\ p)[\alpha]) \\
& \vee\ (\quad \Diamond(\neg\ \text{exists-task-eport}\,(\mathbf{t}))[\alpha] \\
& \qquad ;\ \ \Diamond(\neg\ \text{exists-thread-eport}\,(\mathbf{th}))[\alpha])
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Initialized-Procset} \\
\equiv & \exists\ \mathit{procset} \in \textsc{all-entities}\text{:} \\
& \quad (\quad (\quad \Diamond\text{procset-task-rel}\,(\mathbf{t},\ \mathit{procset})[\alpha] \\
& \qquad\qquad ;\ \ \Diamond\!\uparrow\!\text{procset-thread-rel}\,(\mathbf{th},\ \mathit{procset})[\alpha]) \\
& \qquad \vee\ (\quad \Diamond(\neg\ \text{exists-task-assigned-procset}\,(\mathbf{t}))[\alpha] \\
& \qquad\qquad ;\ \ \Diamond\text{default-procset-rel}\,(\mathit{procset})[\alpha] \\
& \qquad\qquad ;\ \ \Diamond\!\uparrow\!\text{procset-thread-rel}\,(\mathit{procset},\ \mathbf{th})[\alpha]))
\end{array}$$

An *invalid argument* outcome occurs if **t** is found not to be a task. A *failure* outcome is one in which task **t** disappears due to interference from some other agent.

$$\begin{array}{ll}
& \text{Thread-Create-Invalid-Argument} \\
\equiv & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}$$

$$\begin{array}{lll}
& \multicolumn{2}{l}{\text{Thread-Create-Failure}} \\
\equiv & & \Diamond\text{taskp}\,(\mathbf{t})[\alpha] \\
& ; & \Diamond\dagger\text{taskp}\,(\mathbf{t})[\alpha] \\
& ; & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\
& ; & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-failure})[\alpha]
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Create-Resource-Shortage} \\
\equiv & \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-resource-shortage})[\alpha]
\end{array}$$

5.3 thread_get_special_port

DESCRIPTION

Look up one of a thread's special ports.

PARAMETERS

th. The target thread.

which. A scalar value indicating which special port to return.

p. [out] The returned port.

OUTCOMES

There are three possible outcomes: *success*, *failure* and *invalid-argument*.

$$
\begin{array}{ll}
 & \text{Thread-Get-Special-Portp} \\
\equiv & \quad \text{Thread-Get-Special-Port-Success} \\
 & \lor\ \text{Thread-Get-Special-Port-Failure} \\
 & \lor\ \text{Thread-Get-Special-Port-Invalid-Argument}
\end{array}
$$

SPECIFICATION

On a successful outcome, **th** is found to be a thread, and **which** is one of two constants (see below).

$$
\begin{array}{ll}
 & \text{Thread-Get-Special-Port-Success} \\
\equiv & \qquad \text{Thread-Get-Special-Port-Exception-Port} \\
 & \quad \lor\ \text{Thread-Get-Special-Port-Kernel-Port} \\
 & ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}
$$

On a successful outcome, $\mathbf{p} \neq$ NULL-PTR is the port requested by the parameter **which**, while $\mathbf{p}$ =NULL-PTR indicates the thread was found not to have a port in the requested relation.

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Exception-Port} \\
\equiv & \quad \Diamond(\mathbf{which} = \texttt{'exception-port})[\alpha] \\
; & \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\
; & \Diamond\text{Thread-Get-Special-Port-Exception-Port1}
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Exception-Port1} \\
\equiv & \quad (\Diamond(\neg\ \text{exists-thread-eport}\,(\mathbf{th}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{p} = \text{NULL-PTR})[\alpha]) \\
\vee & (\Diamond\text{exists-thread-eport}\,(\mathbf{th})[\alpha]\ ;\ \Diamond\uparrow(\mathbf{p} = \text{thread-eport}\,(\mathbf{th}))[\alpha])
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Kernel-Port} \\
\equiv & \quad \Diamond(\mathbf{which} = \texttt{'kernel-port})[\alpha] \\
; & \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\
; & \Diamond\text{Thread-Get-Special-Port-Kernel-Port1}
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Kernel-Port1} \\
\equiv & \quad (\Diamond(\neg\ \text{exists-thread-sself}\,(\mathbf{th}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{p} = \text{NULL-PTR})[\alpha]) \\
\vee & (\Diamond\text{exists-thread-sself}\,(\mathbf{th})[\alpha]\ ;\ \Diamond\uparrow(\mathbf{p} = \text{thread-sself}\,(\mathbf{th}))[\alpha])
\end{array}$$

An *invalid argument* outcome occurs if **t** is found not to be a task. A *failure* outcome is one in which task **t** disappears due to interference from some other agent.

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Failure} \\
\equiv & \quad \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\
; & \Diamond\dagger\text{threadp}\,(\mathbf{th})[\alpha] \\
; & \Diamond(\neg\ \text{threadp}\,(\mathbf{th}))[\alpha] \\
; & \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-failure})[\alpha]
\end{array}$$

$$\begin{array}{ll}
& \text{Thread-Get-Special-Port-Invalid-Argument} \\
\equiv & \qquad \Diamond(\mathbf{which} \notin \texttt{'(thread-kernel-port} \\
& \qquad\qquad\qquad \texttt{thread-exception-port))}[\alpha] \\
& \quad \vee\ \Diamond(\neg\ \text{threadp}\,(\mathbf{th}))[\alpha] \\
; & \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}$$

5.4 thread_set_special_port

DESCRIPTION

Give a new value to one of a thread's special ports.

PARAMETERS

th. The target thread.

which. A scalar value indicating which special port to set.

p. The new value for the special port, or NULL-PTR.

OUTCOMES

There are three possible outcomes for this program: *success*, *invalid argument* and *failure*.

$$\begin{array}{ll} & \text{Thread-Set-Special-Portp} \\ \equiv & \quad \text{Thread-Set-Special-Port-Success} \\ & \vee\ \text{Thread-Set-Special-Port-Failure} \\ & \vee\ \text{Thread-Set-Special-Port-Invalid-Argument} \end{array}$$

SPECIFICATION

The specification for `thread_set_special_port` is nearly identical to that for `thread_get_special_port`. The only difference is in the interface, where **p** is an input parameter in one, and an output parameter in another.

On a successful outcome, **p** $\neq$NULL-PTR is the new port requested set by the parameter **which**, while **p** =NULL-PTR indicates that no port should be in the requested relation. We say that a special port relation *occurs* ($p[\alpha]$) rather than is *asserted* ($p[\alpha]$), to admit computations that set a special port that was already in that relation.

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Success} \\ \equiv & \quad\quad \text{Thread-Set-Special-Port-Exception-Port} \\ & \quad \vee \ \text{Thread-Set-Special-Port-Kernel-Port} \\ & ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Exception-Port} \\ \equiv & \quad \Diamond(\mathbf{which} = \texttt{'exception-port})[\alpha] \\ & ; \ \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\ & ; \ \Diamond\text{Thread-Set-Special-Port-Exception-Port1} \end{array}$$

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Exception-Port1} \\ \equiv & \quad \Diamond\text{thread-eport-rel}\,(\mathbf{th}, \mathbf{p})[\alpha] \\ & \vee \ (\Diamond(\mathbf{p} = \textsc{null-ptr})[\alpha] \ ; \ \Diamond(\neg \ \text{exists-thread-eport}\,(\mathbf{th}))[\alpha]) \end{array}$$

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Kernel-Port} \\ \equiv & \quad \Diamond(\mathbf{which} = \texttt{'kernel-port})[\alpha] \\ & ; \ \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\ & ; \ \Diamond\text{Thread-Set-Special-Port-Kernel-Port1} \end{array}$$

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Kernel-Port1} \\ \equiv & \quad \Diamond\text{thread-sself-rel}\,(\mathbf{th}, \mathbf{p})[\alpha] \\ & \vee \ (\Diamond(\mathbf{p} = \textsc{null-ptr})[\alpha] \ ; \ \Diamond(\neg \ \text{exists-thread-sself}\,(\mathbf{th}))[\alpha]) \end{array}$$

An *invalid argument* outcome occurs if **t** is found not to be a task. A *failure* outcome is one in which task **t** disappears due to interference from some other agent.

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Failure} \\ \equiv & \quad \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\ & ; \ \Diamond\dagger\text{threadp}\,(\mathbf{th})[\alpha] \\ & ; \ \Diamond(\neg \ \text{threadp}\,(\mathbf{th}))[\alpha] \\ & ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-failure})[\alpha] \end{array}$$

$$\begin{array}{ll} & \text{Thread-Set-Special-Port-Invalid-Argument} \\ \equiv & \quad\quad \Diamond(\mathbf{which} \notin \texttt{'(thread-kernel-port} \\ & \quad\quad\quad\quad\quad\quad \texttt{thread-exception-port))}[\alpha] \\ & \quad \vee \ \Diamond(\neg \ \text{threadp}\,(\mathbf{th}))[\alpha] \\ & ; \ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha] \end{array}$$

5.5 thread_terminate

DESCRIPTION

Deallocate a thread.

PARAMETERS

th. The target thread.

OUTCOMES

There are three possible outcomes: *success*, *invalid argument* and *failure*.

$$\begin{array}{ll} & \text{Thread-Terminatep} \\ \equiv & \quad \text{Thread-Terminate-Success} \\ & \lor\ \text{Thread-Terminate-Invalid-Argument} \\ & \lor\ \text{Thread-Terminate-Failure} \end{array}$$

SPECIFICATION

On a successful return, the thread and its self port are killed.

$$\begin{array}{ll} & \text{Thread-Terminate-Success} \\ \equiv & \quad (\quad \Diamond\text{threadp}\,(\mathbf{th})[\alpha] \\ & \quad\ ;\ \Diamond\forall\, p \in \text{ENTITIES:} \\ & \qquad\qquad (\text{thread-self-rel}\,(\mathbf{th},\, p)[\alpha] \rightarrow \text{Terminate-Port}\,(p)) \\ & \quad\ ;\ \Diamond{\downarrow}\text{entityp}\,(\mathbf{th})[\alpha]) \\ & ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

For thread-terminate, the semantics for the *failure* outcome do not follow the patterm of other kernel calls. It indicates either that the *agent* task or thread is terminated during the execution of the kernel call, or that some other agent interferes with the *target* thread. Depending on which case occurs, **th** may or may not be terminated. We can only infer that the precondition has been met.

An outcome of *invalid-argument* indicates that the argument **th** is not a thread entity.

Thread-Terminate-Failure

$\equiv \Diamond \mathrm{threadp}\,(\mathbf{th})[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]$

Thread-Terminate-Invalid-Argument

$\equiv \Diamond(\neg\ \mathrm{threadp}\,(\mathbf{th}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-failure})[\alpha]$

Chapter 6

Task Interface

6.1 mach_task_self

DESCRIPTION

Look up the sself port for a task. The sself port is usually the same as the self port, unless a debugging task has interposed itself.

PARAMETERS

t. The target task.

p. [out] The task's sself port, or NULL-PTR if no sself port exists.

In the current Mach implementation, a task argument is not supplied to `mach_task_self`; it returns the self port of the currently executing task. The return code is also implicit; it expects the return code to always be `'kern-success`. To be consistent with other specifications, we have included a task **IN** parameter and a return code **OUT** parameter.

OUTCOMES

A successful outcome is the only possibility.

SPECIFICATION

The input argument **t** is found to be a task, and its sself port is found.

$$\begin{array}{ll} & \text{Mach-Task-Selfp} \\ \equiv & \quad \Diamond \text{taskp}\,(\mathbf{t})[\alpha] \\ & ;\ \text{Mach-Task-Self-Found} \\ & ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

If $\mathbf{p} = \text{NULL-PTR}$ is returned, then a state in which **t** has no sself port was encountered. If $\mathbf{p} \neq \text{NULL-PTR}$ is returned, then a state in which **p** is **t**'s sself port was encountered.

$$\begin{array}{ll} & \text{Mach-Task-Self-Found} \\ \equiv & \quad (\Diamond(\neg\ \text{exists-task-sself}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{p} = \text{NULL-PTR})[\alpha]) \\ & \vee\ (\Diamond \text{exists-task-sself}\,(\mathbf{t})[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{p} = \text{task-sself}\,(\mathbf{t}))[\alpha]) \end{array}$$

6.2 task_create

DESCRIPTION

Create a new (child) task using an existing (parent) task as a template.

PARAMETERS

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

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

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

OUTCOMES

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

$$\begin{array}{ll} & \text{Task-Createp} \\ \equiv & \quad \text{Task-Create-Success} \\ & \vee\ \text{Task-Create-Invalid-Arg} \\ & \vee\ \text{Task-Create-Resource-Shortage} \end{array}$$

SPECIFICATION

On a successful outcome, $\mathbf{t_1}$ is found to be a task, and the child task is created and initialized. Initialization includes creation of several special ports, and inheritance of the parent's address space and processor set. We specify nothing about the resource shortage outcome.

$$\begin{array}{ll} & \text{Task-Create-Success} \\ \equiv & \quad \Diamond\text{taskp}\,(\mathbf{t_1})[\alpha] \\ & ;\ \Diamond\uparrow\text{taskp}\,(\mathbf{t_2})[\alpha] \\ & ;\ \text{Task-Initialized} \\ & ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

$$
\begin{array}{ll}
& \text{Task-Initialized} \\
\equiv & \quad \text{Task-Self-Created} \\
& \wedge\ \text{Task-Bport-Initialized} \\
& \wedge\ \text{Task-Eport-Initialized} \\
& \wedge\ (\textbf{inh-flg}[\texttt{'alpha}] \rightarrow \text{Task-Memory-Inherited}) \\
& \wedge\ ((\neg\ \textbf{inh-flg})[\texttt{'alpha}] \rightarrow \text{Task-Memory-Not-Inherited}) \\
& \wedge\ \text{Task-Procset-Inherited}
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

$$
\begin{array}{l}
\quad \text{Task-Memory-Inherited} \\
\equiv \forall\, \mathtt{0} \le vpa < \textsc{address-space-limit}: \\
\qquad (\quad \text{page-aligned}\,(vpa)[\alpha] \\
\qquad\ \ \rightarrow \quad (\quad \Diamond(\neg\ \text{allocated}\,(\mathbf{t_1},\ vpa))[\alpha] \\
\qquad\qquad\qquad ;\ \ \Diamond(\neg\ \text{allocated}\,(\mathbf{t_2},\ vpa))[\alpha]) \\
\qquad\qquad \lor\ \text{Task-Memory-None}\,(vpa) \\
\qquad\qquad \lor\ \text{Task-Memory-Share}\,(vpa) \\
\qquad\qquad \lor\ \text{Task-Memory-Copy}\,(vpa))
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Task-Memory-None}\,(vpa) \\
\equiv \exists\, m \in \textsc{all-entities},\ \mathtt{0} \le o < \textsc{memorysize},\ cp \in \textsc{all-psets}, \\
\qquad mp \in \textsc{all-psets}: \\
\qquad (\quad \Diamond\text{map-rel}\,(\mathbf{t_1},\ m,\ vpa,\ o,\ \mathtt{'none},\ cp,\ mp)[\alpha] \\
\qquad\ ;\ \ \Diamond(\neg\ \text{allocated}\,(\mathbf{t_2},\ vpa))[\alpha])
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Task-Memory-Share}\,(vpa) \\
\equiv \exists\, m \in \textsc{all-entities},\ \mathtt{0} \le o < \textsc{memorysize},\ cp \in \textsc{all-psets}, \\
\qquad mp \in \textsc{all-psets}: \\
\qquad (\quad \Diamond\text{map-rel}\,(\mathbf{t_1},\ m,\ vpa,\ o,\ \mathtt{'share},\ cp,\ mp)[\alpha] \\
\qquad\ ;\ \ \Diamond\uparrow\text{map-rel}\,(\mathbf{t_2},\ m,\ vpa,\ o,\ \mathtt{'share},\ cp,\ mp)[\alpha])
\end{array}
$$

Task-Memory-Copy (vpa)

$$\begin{array}{l}
\equiv \exists\, m_1 \in \text{ALL-ENTITIES},\ \mathtt{0} \le o_1 < \text{MEMORYSIZE},\ cp \in \text{ALL-PSETS}, \\
\quad mp \in \text{ALL-PSETS}: \\
\quad (\quad \Diamond\text{map-rel}\,(\mathbf{t_1},\ m_1,\ vpa,\ o_1,\ \mathtt{'copy},\ cp,\ mp)[\alpha] \\
\quad\ ;\ \exists\, m_2 \in \text{ALL-ENTITIES},\ \mathtt{0} \le o_2 < \text{MEMORYSIZE}: \\
\qquad (\quad \Diamond\!\uparrow\!\text{memoryp}\,(m_2)[\alpha] \\
\qquad\ ;\quad \Diamond\!\uparrow\!\text{temporary-rel}\,(m_2)[\alpha] \\
\qquad\quad \wedge\ \text{Extract-Va-Region}\,(\mathbf{t_1},\ vpa,\ o_2,\ \text{PAGESIZE},\ m_2) \\
\qquad\ ;\ \Diamond\!\uparrow\!\text{map-rel}\,(\mathbf{t_2},\ m_2,\ vpa,\ \mathtt{'o2},\ \mathtt{'copy},\ cp,\ mp)[\alpha]))
\end{array}$$

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

Task-Memory-Not-Inherited

$$\equiv \forall\ \mathtt{0} \le va < \text{ADDRESS-SPACE-LIMIT}:\ (\Diamond(\neg\ \text{allocated}\,(\mathbf{t_2},\ va))[\alpha])$$

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

Task-Procset-Inherited

$$\begin{array}{l}
\equiv \qquad \exists\, procset \in \text{ALL-ENTITIES}: \\
\qquad\quad (\quad \Diamond\text{procset-task-rel}\,(procset,\ \mathbf{t_1})[\alpha] \\
\qquad\quad\ ;\ \Diamond\!\uparrow\!\text{procset-task-rel}\,(procset,\ \mathbf{t_2})[\alpha]) \\
\quad \vee\ (\quad \Diamond(\neg\ \text{exists-task-assigned-procset}\,(\mathbf{t_1}))[\alpha] \\
\qquad\ ;\ \exists\, procset \in \text{ALL-ENTITIES}: \\
\qquad\quad (\quad \Diamond\text{default-procset-rel}\,(procset)[\alpha] \\
\qquad\quad\ ;\ \Diamond\!\uparrow\!\text{procset-task-rel}\,(procset,\ \mathbf{t_2})[\alpha]))
\end{array}$$

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

Task-Create-Invalid-Arg

$$\equiv \Diamond(\neg\ \text{taskp}\,(\mathbf{t_1}))[\alpha]\ ;\ \Diamond\!\uparrow\!(\mathbf{rc} = \mathtt{'kern\text{-}invalid\text{-}arg})[\alpha]$$

Task-Create-Resource-Shortage

$$\equiv \Diamond\!\uparrow\!(\mathbf{rc} = \mathtt{'kern\text{-}resource\text{-}shortage})[\alpha]$$

6.3 task_get_special_port

DESCRIPTION

Look up one of a task's special ports.

PARAMETERS

t. The target task.

which. A scalar value indicating which special port to return.

p. [out] The returned port.

OUTCOMES

There are three possible outcomes: *success*, *failure* and *invalid-argument.*

$$\begin{array}{ll} & \text{Task-Get-Special-Portp} \\ \equiv & \quad \text{Task-Get-Special-Port-Success} \\ & \vee\ \text{Task-Get-Special-Port-Failure} \\ & \vee\ \text{Task-Get-Special-Port-Invalid-Arg} \end{array}$$

SPECIFICATION

On a successful outcome, **t** is found to be a task, and **which** is one of three constants (see below). An *invalid argument* outcome occurs if **t** is found not to be a task, or **which** has a bad value. A *failure* outcome is one in which task **t** disappears due to interference from some other process.

$$\begin{array}{ll} & \text{Task-Get-Special-Port-Failure} \\ \equiv & \quad \Diamond \text{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond \dagger \text{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond (\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\ ; & \Diamond \uparrow (\mathbf{rc} = \texttt{'kern-failure})[\alpha] \end{array}$$

$$\begin{array}{ll}
& \text{Task-Get-Special-Port-Invalid-Arg} \\
\equiv & \qquad \Diamond(\mathbf{which} \notin \texttt{'(task-bootstrap-port task-exception-port} \\
& \qquad\qquad\qquad \texttt{task-kernel-port))}[\alpha] \\
& \quad \vee \ \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\
& \ ; \ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{array}$$

$$\begin{array}{ll}
& \text{Task-Get-Special-Port-Success} \\
\equiv & \qquad \text{Task-Get-Special-Portp-Bootstrap} \\
& \quad \vee \ \text{Task-Get-Special-Portp-Exception} \\
& \quad \vee \ \text{Task-Get-Special-Kernel-Port-Success} \\
& \ ; \ \Diamond\!\uparrow\!(\mathbf{rc} = \texttt{'kern-success})[\alpha]
\end{array}$$

On a successful outcome, $\mathbf{p} \neq$NULL-PTR is the port requested by the parameter **which**, while $\mathbf{p}$ =NULL-PTR indicates the task was found not to have a port in the requested relation.

$$\begin{array}{ll}
& \text{Task-Get-Special-Portp-Bootstrap} \\
\equiv & \ \Diamond(\mathbf{which} = \texttt{'task-bootstrap-port})[\alpha] \\
& \ ; \quad (\Diamond(\neg\ \text{exists-task-bport}\,(\mathbf{t}))[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \textsc{null-ptr})[\alpha]) \\
& \quad \vee \ (\Diamond\text{exists-task-bport}\,(\mathbf{t})[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \text{task-bport}\,(\mathbf{t}))[\alpha])
\end{array}$$

$$\begin{array}{ll}
& \text{Task-Get-Special-Portp-Exception} \\
\equiv & \ \Diamond(\mathbf{which} = \texttt{'task-exception-port})[\alpha] \\
& \ ; \quad (\Diamond(\neg\ \text{exists-task-eport}\,(\mathbf{t}))[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \textsc{null-ptr})[\alpha]) \\
& \quad \vee \ (\Diamond\text{exists-task-eport}\,(\mathbf{t})[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \text{task-eport}\,(\mathbf{t}))[\alpha])
\end{array}$$

$$\begin{array}{ll}
& \text{Task-Get-Special-Kernel-Port-Success} \\
\equiv & \ \Diamond(\mathbf{which} = \texttt{'task-kernel-port})[\alpha] \\
& \ ; \quad (\Diamond(\neg\ \text{exists-task-sself}\,(\mathbf{t}))[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \textsc{null-ptr})[\alpha]) \\
& \quad \vee \ (\Diamond\text{exists-task-sself}\,(\mathbf{t})[\alpha] \ ; \ \Diamond\!\uparrow\!(\mathbf{p} = \text{task-sself}\,(\mathbf{t}))[\alpha])
\end{array}$$

6.4 task_set_special_port

DESCRIPTION

Set one of a task's special ports: the sself port, the bootstrap port, or the exception port.

PARAMETERS

t. The target task.

p. The special port, or NULL-PTR.

which. A scalar value indicating which special port to set.

OUTCOMES

There are three possible outcomes for this program: *success*, *invalid argument* and *failure*.

$$\begin{array}{ll} & \text{Task-Set-Special-Portp} \\ \equiv & \quad \text{Task-Set-Special-Port-Success} \\ & \lor \ \text{Task-Set-Special-Port-Failure} \\ & \lor \ \text{Task-Set-Special-Port-Invalid-Arg} \end{array}$$

SPECIFICATION

The specification for `task_set_special_port` is nearly identical to that for `task_get_special_port`. The only difference is in the interface, where **p** is an input parameter in one, and an output parameter in another.

On a successful outcome, **t** is found to be a task, and **which** is one of three constants (see below). An *invalid argument* outcome occurs if **t** is found not to be a task, or if **which** has a bad value. A failure outcome is one in which task **t** disappears due to interference from some other process.

$$\begin{array}{ll} & \text{Task-Set-Special-Port-Failure} \\ \equiv & \Diamond\text{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond\dagger\text{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\ ; & \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-failure})[\alpha] \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Task-Set-Special-Port-Invalid-Arg}} \\ \equiv & & \Diamond(\mathbf{which} \notin \texttt{'(task-bootstrap-port task-exception-port} \\ & & \qquad\qquad \texttt{task-kernel-port)})[\alpha] \\ & \vee & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\ ; & \multicolumn{2}{l}{\Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]} \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Task-Set-Special-Port-Success}} \\ \equiv & & \text{Task-Set-Special-Bootstrap-Port-Success} \\ & \vee & \text{Task-Set-Special-Portp-Exception} \\ & \vee & \text{Task-Set-Special-Kernel-Port-Success} \\ ; & \multicolumn{2}{l}{\Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha]} \end{array}$$

On a successful outcome, $\mathbf{p} \neq$NULL-PTR is the new port requested set by the parameter *which*, while $\mathbf{p} =$NULL-PTR indicates that no port should be in the requested relation.

We say that a special port relation *occurs* ($p[\alpha]$) rather than is *asserted* ($\uparrow p[\alpha]$), to admit computations that set a special port that was already in that relation.

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Task-Set-Special-Bootstrap-Port-Success}} \\ \equiv & \multicolumn{2}{l}{\Diamond(\mathbf{which} = \texttt{'task-bootstrap-port})[\alpha]} \\ ; & & (\Diamond(\mathbf{p} = \text{NULL-PTR})[\alpha]\ ;\ \Diamond(\neg\ \text{exists-task-bport}\,(\mathbf{t}))[\alpha]) \\ & \vee & \Diamond\text{task-bport-rel}\,(\mathbf{t},\ \mathbf{p})[\alpha] \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Task-Set-Special-Portp-Exception}} \\ \equiv & \multicolumn{2}{l}{\Diamond(\mathbf{which} = \texttt{'task-exception-port})[\alpha]} \\ ; & & (\Diamond(\mathbf{p} = \text{NULL-PTR})[\alpha]\ ;\ \Diamond(\neg\ \text{exists-task-eport}\,(\mathbf{t}))[\alpha]) \\ & \vee & \Diamond\text{task-eport-rel}\,(\mathbf{t},\ \mathbf{p})[\alpha] \end{array}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{\text{Task-Set-Special-Kernel-Port-Success}} \\ \equiv & \multicolumn{2}{l}{\Diamond(\mathbf{which} = \texttt{'task-kernel-port})[\alpha]} \\ ; & & (\Diamond(\mathbf{p} = \text{NULL-PTR})[\alpha]\ ;\ \Diamond(\neg\ \text{exists-task-sself}\,(\mathbf{t}))[\alpha]) \\ & \vee & \Diamond\text{task-sself-rel}\,(\mathbf{t},\ \mathbf{p})[\alpha] \end{array}$$

6.5 task_terminate

DESCRIPTION

Kill a task and free its resources.

PARAMETERS

t. The target task.

OUTCOMES

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

$$\begin{aligned} & \text{Task-Terminate}_p \\ \equiv\ & \quad \text{Task-Terminate-Success} \\ & \lor\ \text{Task-Terminate-Failure} \\ & \lor\ \text{Task-Terminate-Invalid-Arg} \end{aligned}$$

SPECIFICATION

On a successful outcome, the task and its resources have been removed. When the task is successfully terminated, a number of other entities are terminated as well. The task's self port, if it exists, is terminated. All of the task's threads are terminated. All of the ports to which the task has a receive right are terminated. All of the memories which are mapped into the target task, but into no other task, are terminated.

For a discussion of the recursive nature of entity destruction in Mach, see Section 7.4.1.

$$
\begin{aligned}
& \text{Task-Terminate-Success} \\
\equiv \quad & (\quad \Diamond \text{taskp}\,(\mathbf{t})[\alpha] \\
& \quad ; \quad \Diamond \forall\, p \in \textsc{entities}: \\
& \qquad\qquad (\text{task-self-rel}\,(\mathbf{t},\, p)[\alpha] \rightarrow \text{Terminate-Port}\,(p)) \\
& \qquad \wedge\ \Diamond \forall\, th \in \text{threads}\,(\mathbf{t}):\ \text{Terminate-Thread}\,(th) \\
& \qquad \wedge\ \Diamond \forall\, n \in \mathcal{N},\, p \in \textsc{all-entities}: \\
& \qquad\qquad (\quad \text{r-right}\,(\mathbf{t},\, n)[\alpha] \wedge (\text{named-port}\,(\mathbf{t},\, n) = p)[\alpha] \\
& \qquad\qquad\ \rightarrow \text{Terminate-Port}\,(p)) \\
& \qquad \wedge\ \Diamond \forall\, m \in \textsc{entities}: \\
& \qquad\qquad (\quad (\text{mapping-tasks}\,(m) = \{\mathbf{t}\})[\alpha] \\
& \qquad\qquad\ \rightarrow \text{Terminate-Memory}\,(m)) \\
& \quad ; \ \Diamond{\downarrow}\text{entityp}\,(\mathbf{t})[\alpha]) \\
& ; \ \Diamond(\Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-success})[\alpha])
\end{aligned}
$$

The *invalid argument* outcome occurs when **t** is found not to be a task. A *failure* outcome is one in which the task **t** disappears due to interference from some other agent.

$$
\begin{aligned}
& \text{Task-Terminate-Invalid-Arg} \\
\equiv\ & \Diamond(\neg\ \text{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond{\uparrow}(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]
\end{aligned}
$$

$$
\begin{aligned}
& \text{Task-Terminate-Failure} \\
\equiv \quad & \Diamond \text{taskp}\,(\mathbf{t})[\alpha] \\
& ;\ \Diamond \dagger \text{taskp}\,(\mathbf{t})[\alpha] \\
& ;\ (\neg\ \text{taskp}\,(\mathbf{t}))[\alpha] \\
& ;\ {\uparrow}(\mathbf{rc} = \texttt{'kern-failure})[\alpha]
\end{aligned}
$$

6.6 task_threads

DESCRIPTION

Return the threads associated with a task.

PARAMETERS

t. The target task.

threads. [out] The returned list of threads.

OUTCOMES

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

$$\begin{array}{ll} & \text{Task-Threadsp} \\ \equiv & \quad \text{Task-Threadsp-Success} \\ & \vee\ \text{Task-Threadsp-Invalid-Argument} \\ & \vee\ \text{Task-Threadsp-Failure} \\ & \vee\ \text{Task-Threads-Resource-Shortage} \end{array}$$

SPECIFICATION

On a successful outcome, the **threads** argument represents a snapshot of the task's threads.

$$\begin{array}{ll} & \text{Task-Threadsp-Success} \\ \equiv & \quad \Diamond(\text{taskp}\,(\mathbf{t}) \wedge (\mathbf{threads} = \text{threads}\,(\mathbf{t})))[\alpha] \\ & ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-success})[\alpha] \end{array}$$

A failure outcome identifies a computation in which the task of interest disappears due to interference by some other agent. The *invalid-argument* outcome occurs when the argument **t** is not a task in some state in the computation[1].

[1] The distinction between these two outcomes in the implementation is the point in the computation when the check is made.

Task-Threadsp-Failure

$$\begin{array}{rl} \equiv & \Diamond \mathrm{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond \dagger \mathrm{taskp}\,(\mathbf{t})[\alpha] \\ ; & \Diamond(\neg\ \mathrm{taskp}\,(\mathbf{t}))[\alpha] \\ ; & \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-failure})[\alpha] \end{array}$$

Task-Threadsp-Invalid-Argument

$$\equiv \Diamond(\neg\ \mathrm{taskp}\,(\mathbf{t}))[\alpha]\ ;\ \Diamond\uparrow(\mathbf{rc} = \texttt{'kern-invalid-argument})[\alpha]$$

We do not specify anything about the computation in the case of *resource shortage*.

Task-Threads-Resource-Shortage

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

Chapter 7

Common Specifications

7.1 Introduction

This chapter presents specifications for some of the lower-level actions made on kernel resources. These are used throughout the kernel interface specification. Section 7.2 presents specifications for operations on local port names. Section 7.3 gives virtual memory specifications, and Section 7.4 discusses the termination of kernel entities.

7.2 Actions on Local Names

Local names can name port rights, dead rights, and port sets. Port rights can be further subdivided into send, receive, and send once rights. A local name can name multiple send rights to a port, or multiple dead rights. A reference count associated with each local name gives the number of times the name has been assigned.

It is common usage to refer to a single reference by a local name as a "right". A local name can reference a single receive or send-once right, or a number of send rights, or a number of dead name "rights", or a single port set "right". For brevity in this section, we will follow this convention.

The fundamental actions on local names are allocating or deallocating individual rights. Allocating the first right reserves a new local name. For send/receive rights and dead rights, allocating additional rights for a local name simply increases the reference count. Also, send and receive rights to a port coalesce under a single name. For receive rights, send-once rights and port sets, allocating additional rights is not allowed.

Related actions are *inserting* and *extracting* rights. Conceptually, allocating and deallocating rights create and destroy the rights, while extracting and inserting are involved with moving the rights from one place to another. Inserting a right simply allocates the right. Extracting a right from a local name may deallocate the right or clone a new right (leaving the local name unchanged).

7.2.1 Allocating Rights

This section specifies behaviors of the kernel when a port right of a certain type is allocated for a local name. When an additional right is allocated for an existing send/receive right or dead name, the reference count is incremented, up to a maximum value. The specifications for send rights and dead rights have an extra parameter δ, which allows several rights to be allocated at once. For most invocations, $\delta = \mathtt{1}$.

We specify two cases: either a *new* local name is created, or the right is *coalesced* with an existing name. Additionally, we create composite *allocate* specifications for use where the distinction is not important.

Creating New Rights

The predicates in this section specify the situation where a new local name is reserved for a given right. A new send or receive right for a port cannot be allocated to a name if the port already has a send or receive right by a different name.

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{New-Send-Right}\,(t,\ p,\ n,\ \delta)} \\
\equiv & & \Diamond(\neg\ \text{local-namep}\,(t,\ n))[\alpha] \\
 & \wedge & \Diamond(\delta < \text{MAX-REFCOUNT})[\alpha] \\
 & \wedge & \Diamond\neg\ \exists\, n' \in \mathcal{N}\!: \\
 & & \quad (\quad (\text{s-right}\,(t,\ n')[\alpha] \vee \text{r-right}\,(t,\ n')[\alpha]) \\
 & & \quad\quad \wedge\ (p = \text{named-port}\,(t,\ n'))[\alpha]) \\
 ; & \multicolumn{2}{l}{\Diamond\!\uparrow\!\text{port-right-rel}\,(t,\ p,\ n,\ \{\mathtt{'send}\},\ \delta)[\alpha]}
\end{array}
$$

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\text{New-Receive-Right}\,(t,\ p,\ n)} \\
\equiv & & \Diamond\neg\ \exists\, n' \in \mathcal{N}\!: \\
 & & \quad (\quad (\text{s-right}\,(t,\ n')[\alpha] \vee \text{r-right}\,(t,\ n')[\alpha]) \\
 & & \quad\quad \wedge\ (p = \text{named-port}\,(t,\ n'))[\alpha]) \\
 & \wedge & \Diamond(\neg\ \text{local-namep}\,(t,\ n))[\alpha] \\
 ; & \multicolumn{2}{l}{\Diamond\!\uparrow\!\text{port-right-rel}\,(t,\ p,\ n,\ \{\mathtt{'receive}\},\ \mathtt{1})[\alpha]}
\end{array}
$$

$$
\begin{array}{ll}
 & \text{New-Send-Once-Right}\,(t,\ p,\ n) \\
\equiv & \Diamond(\neg\ \text{local-namep}\,(t,\ n))[\alpha] \\
 ; & \Diamond\!\uparrow\!\text{port-right-rel}\,(t,\ p,\ n,\ \{\mathtt{'send\text{-}once}\},\ \mathtt{1})[\alpha]
\end{array}
$$

New-Dead-Right $(t,\ n,\ \delta)$
$\equiv \quad \Diamond(\neg\ \text{local-namep}\,(t,\ n))[\alpha] \land \Diamond(\delta < \textsc{max-refcount})[\alpha]$
$\quad ;\ \Diamond{\uparrow}\text{dead-right-rel}\,(t,\ n,\ \delta)[\alpha]$

New-Port-Set $(t,\ n)$
$\equiv \Diamond(\neg\ \text{local-namep}\,(t,\ n))[\alpha]\ ;\ \Diamond{\uparrow}\text{port-set-rel}\,(t,\ n,\ \emptyset)[\alpha]$

Coalescing Rights

If a send right to a port is allocated for a local name space where the port already has a send or receive right, then the reference count is incremented. We divide the specification into four cases: a send right can coalesce with a local name that represents send, receive, or both send and receive rights. Also, dead rights for a given name coalesce.

Coalesce-Send-Send-Right $(t,\ p,\ n,\ \delta)$
$\equiv \exists\ 1 \leq j < \textsc{max-refcount}:$
$\quad (\quad \Diamond(\quad \text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ j)[\alpha]$
$\qquad\qquad \land\ (j + \delta \leq \textsc{max-refcount})[\alpha])$
$\qquad ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ j + \delta)[\alpha])$

Coalesce-Send-Receive-Right $(t,\ p,\ n,\ \delta)$
$\equiv (\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'receive}\},\ 1)[\alpha]$
$\quad ;\ \Diamond(\delta < \textsc{max-refcount})[\alpha]$
$\quad ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ \delta + 1)[\alpha])$

Coalesce-Send-Sr-Right $(t,\ p,\ n,\ \delta)$
$\equiv \exists\ 1 \leq j < \textsc{max-refcount}:$
$\quad (\quad \Diamond(\quad \text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ j)[\alpha]$
$\qquad\qquad \land\ (j + \delta \leq \textsc{max-refcount})[\alpha])$
$\qquad ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ j + \delta)[\alpha])$

Coalesce-Receive-Send-Right $(t,\ p,\ n)$
$\equiv \exists\ 1 \leq j < (\textsc{max-refcount} - 1):$
$\quad (\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ j)[\alpha]$
$\qquad ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ j + 1)[\alpha])$

Coalesce-Dead-Right $(t,\ n,\ \delta)$
$\equiv \exists\ 1 \leq i < \textsc{max-refcount}:$
$\quad (\quad \Diamond(\text{dead-right-rel}\,(t,\ n,\ i)[\alpha] \land (i + \delta \leq \textsc{max-refcount})[\alpha])$
$\qquad ;\ \Diamond{\uparrow}\text{dead-right-rel}\,(t,\ n,\ i + \delta)[\alpha])$

Composite Specifications

In most contexts, the distinctions among the different ways to allocate rights are not important. We provide composite specifications for these cases.

$$\begin{aligned}
&\text{Allocate-Send-Right}\,(t,\ p,\ n,\ \delta)\\
\equiv\ &\quad\ \ \text{New-Send-Right}\,(t,\ p,\ n,\ \delta)\\
&\ \vee\ \text{Coalesce-Send-Send-Right}\,(t,\ p,\ n,\ \delta)\\
&\ \vee\ \text{Coalesce-Send-Receive-Right}\,(t,\ p,\ n,\ \delta)\\
&\ \vee\ \text{Coalesce-Send-Sr-Right}\,(t,\ p,\ n,\ \delta)
\end{aligned}$$

$$\begin{aligned}
&\text{Coalesce-Send-Right}\,(t,\ p,\ n,\ \delta)\\
\equiv\ &\quad\ \ \text{Coalesce-Send-Send-Right}\,(t,\ p,\ n,\ \delta)\\
&\ \vee\ \text{Coalesce-Send-Receive-Right}\,(t,\ p,\ n,\ \delta)\\
&\ \vee\ \text{Coalesce-Send-Sr-Right}\,(t,\ p,\ n,\ \delta)
\end{aligned}$$

$$\begin{aligned}
&\text{Allocate-Receive-Right}\,(t,\ p,\ n)\\
\equiv\ &\text{New-Receive-Right}\,(t,\ p,\ n) \vee \text{Coalesce-Receive-Send-Right}\,(t,\ p,\ n)
\end{aligned}$$

$$\begin{aligned}
&\text{Allocate-Dead-Right}\,(t,\ n,\ \delta)\\
\equiv\ &\text{New-Dead-Right}\,(t,\ n,\ \delta) \vee \text{Coalesce-Dead-Right}\,(t,\ n,\ \delta)
\end{aligned}$$

7.2.2 Deallocating Port Rights

In this section, we specify behaviors in which rights are deallocated from a local name. A user task can make this request directly as specified in Mach-Port-Mod-Refsp and others, or indirectly as in Mach-Msg-Sendp. The argument δ is the number of rights of this type to deallocate — the amount by which the reference count is decremented. In the typical case, $\delta = \texttt{1}$.

As for allocating rights, we specify two classes of behaviors: either the local name is *removed* from the task's port name space, or the rights are *de-coalesced* from additional rights. Additionally, we provide composite *deallocate* specifications for the instances where the distinction is not important.

Removing Local Names

In this section we specify behaviors that cause a local name to be removed from a task's name space.

For a send-only right, the local name is deallocated from the name space of the target task if the current reference count becomes zero[1].

$$\begin{aligned}&\text{Remove-Send-Right}\,(t,\ n,\ \delta)\\ \equiv\ &\exists\ p \in \text{ALL-ENTITIES:}\\ &\quad(\Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ \delta)[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha])\end{aligned}$$

$$\begin{aligned}&\text{Remove-Receive-Right}\,(t,\ n)\\ \equiv\ &\exists\ p \in \text{ALL-ENTITIES:}\\ &\quad(\Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'receive}\},\ \texttt{1})[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha])\end{aligned}$$

A send-once right always has a reference count of 1, so decrementing the reference count causes the name to be deallocated from the target task's name space[2].

$$\text{Remove-Send-Once-Right}\,(t,\ n) \equiv \Diamond\text{so-right}\,(t,\ n)[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha]$$

$$\begin{aligned}&\text{Remove-Dead-Right}\,(t,\ n,\ \delta)\\ \equiv\ &\Diamond\text{dead-right-rel}\,(t,\ n,\ \delta)[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha]\end{aligned}$$

$$\begin{aligned}&\text{Remove-Port-Set-Name}\,(t,\ n)\\ \equiv\ &\Diamond\text{port-set-namep}\,(t,\ n)[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha]\end{aligned}$$

De-Coalescing Rights

Send rights coalesce with receive rights. If a send right shares a local name with a receive right, the receive right contributes one to the reference count. A send-receive right is converted to a receive-only right when the reference count becomes one.

[1] A no-more-senders notification is sent if the receiver requests it, and the last send right to the port is deleted. We do not model this activity.

[2] A temporal invariant of the kernel is that eventually exactly one message is queued for each send-once right created for a port. If a send-once right is destroyed, the kernel enqueues a *send-once notification* message alerting the receiver of that occurrence. We do not model this activity.

$$\begin{aligned}
&\text{De-Coalesce-Send-Send-Right}\,(t,\ n,\ \delta)\\
\equiv\ &\exists\, p \in \textsc{all-entities},\ \mathtt{1} \le i < \textsc{max-refcount}:\\
&\quad(\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ i)[\alpha]\\
&\quad\ \ ;\ \Diamond(\delta < i)[\alpha]\\
&\quad\ \ ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ i - \delta)[\alpha])
\end{aligned}$$

$$\begin{aligned}
&\text{De-Coalesce-Send-Sr-Right}\,(t,\ n,\ \delta)\\
\equiv\ &\exists\, p \in \textsc{all-entities},\ \mathtt{1} \le i < \textsc{max-refcount}:\\
&\quad(\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ i)[\alpha]\\
&\quad\ \ ;\ \Diamond(\delta < i)[\alpha]\\
&\quad\ \ ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ i - \delta)[\alpha])
\end{aligned}$$

When a receive right is destroyed, remaining send rights are turned into dead names.

$$\begin{aligned}
&\text{De-Coalesce-Receive-Sr-Right}\,(t,\ n)\\
\equiv\ &\exists\, p \in \textsc{all-entities},\ \mathtt{1} \le i < \textsc{max-refcount}:\\
&\quad(\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send},\ \texttt{'receive}\},\ i)[\alpha]\\
&\quad\ \ ;\ \Diamond{\uparrow}\text{dead-right-rel}\,(t,\ n,\ i - \mathtt{1})[\alpha])
\end{aligned}$$

$$\begin{aligned}
&\text{De-Coalesce-Dead-Right}\,(t,\ n,\ \delta)\\
\equiv\ &\exists\, \mathtt{1} \le i < \textsc{max-refcount}:\\
&\quad(\quad \Diamond\text{dead-right-rel}\,(t,\ n,\ i)[\alpha]\\
&\quad\ \ ;\ \Diamond(\delta < i)[\alpha]\\
&\quad\ \ ;\ \Diamond{\uparrow}\text{dead-right-rel}\,(t,\ n,\ i - \mathtt{1})[\alpha])
\end{aligned}$$

Composite Specifications

In mny cases, the distinctions amoung the different ways to remove or de-coalesce rights is not important. We provide composite specifications for thse cases.

$$\begin{aligned}
&\text{Deallocate-Send-Right}\,(t,\ n,\ \delta)\\
\equiv\ &\quad\ \text{Remove-Send-Right}\,(t,\ n,\ \delta)\\
&\ \vee\ \text{De-Coalesce-Send-Send-Right}\,(t,\ n,\ \delta)\\
&\ \vee\ \text{De-Coalesce-Send-Sr-Right}\,(t,\ n,\ \delta)
\end{aligned}$$

$$\begin{aligned}
&\text{Deallocate-Receive-Right}\,(t,\ n)\\
\equiv\ &\text{Remove-Receive-Right}\,(t,\ n) \vee \text{De-Coalesce-Receive-Sr-Right}\,(t,\ n)
\end{aligned}$$

$$\begin{aligned}&\text{Deallocate-Dead-Right}\,(t,\ n,\ \delta)\\ \equiv\ &\text{Remove-Dead-Right}\,(t,\ n,\ \delta) \vee \text{De-Coalesce-Dead-Right}\,(t,\ n,\ \delta)\end{aligned}$$

7.2.3 Moving Port Rights

The action of moving a port right from one task to another is decomposed into an *extraction* (from the source task) followed by an *insertion* (into the destination task). When a port right is moved via IPC, there is an intermediate state in which the right is a *transit right* encoded in a queued message.

Extracting a right is closely related, but not equivalent, to deallocating a right (Section 7.2.2). Extracting a right from a task's local name space need not cause a change in the name space – extracted rights can be cloned from existing rights. A right can be moved from a name space, in which case the effects are seen in the name space.

Preconditions for Right Movement

A pair $\langle instr,\ r\rangle$ describes the extraction of a right. Six combinations of right r and instruction *instr* are legal for extracting a right, as enumerated in the definitions below. For the special case of creating transit rights, a dead local name may be substituted for a send or send-once right if the instruction is `'move` or `'copy`. The second definition describes the case where a dead name is allowed.

A pair $\langle instr, right\rangle$ specifies a right to moved and the method by which it is moved. These are the legal pairs.

$$\begin{aligned}&\text{Legal-Instr-Right}\,(instr,\ r)\\ \equiv\ &\quad \Diamond(\langle instr,\ r\rangle = \langle \texttt{'move},\ \texttt{'receive}\rangle)[\alpha]\\ &\vee\ \Diamond(\langle instr,\ r\rangle = \langle \texttt{'move},\ \texttt{'send}\rangle)[\alpha]\\ &\vee\ \Diamond(\langle instr,\ r\rangle = \langle \texttt{'move},\ \texttt{'send-once}\rangle)[\alpha]\\ &\vee\ \Diamond(\langle instr,\ r\rangle = \langle \texttt{'copy},\ \texttt{'send}\rangle)[\alpha]\\ &\vee\ \Diamond(\langle instr,\ r\rangle = \langle \texttt{'make},\ \texttt{'send}\rangle)[\alpha]\\ &\vee\ \Diamond(\langle instr,\ r\rangle = \langle \texttt{'make},\ \texttt{'send-once}\rangle)[\alpha]\end{aligned}$$

The following specification recognizes a computation in which α recognizes that a target task $\mathbf{t}$ has the correct right for a given $\langle instr, right\rangle$ pair.

$$
\begin{array}{ll}
 & \text{Legal-Name-Instr-Right}\,(t,\ n,\ \mathit{instr},\ r) \\
\equiv & \quad \Diamond((\langle \mathit{instr},\ r\rangle = \langle \texttt{'move, 'receive}\rangle)[\alpha] \wedge \text{r-right}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond((\langle \mathit{instr},\ r\rangle = \langle \texttt{'move, 'send}\rangle)[\alpha] \wedge \text{s-right}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond(\quad (\langle \mathit{instr},\ r\rangle = \langle \texttt{'move, 'send-once}\rangle)[\alpha] \\
 & \qquad \wedge\ \text{so-right}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond((\langle \mathit{instr},\ r\rangle = \langle \texttt{'copy, 'send}\rangle)[\alpha] \wedge \text{s-right}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond((\langle \mathit{instr},\ r\rangle = \langle \texttt{'make, 'send}\rangle)[\alpha] \wedge \text{r-right}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond((\langle \mathit{instr},\ r\rangle = \langle \texttt{'make, 'send-once}\rangle)[\alpha] \wedge \text{r-right}\,(t,\ n)[\alpha])
\end{array}
$$

The predicate Legal-Name-Instr-Right-Deadok describes a computation in which a target task **t** has either the correct preconditions for a given $\langle \mathit{instr}, \mathit{right}\rangle$ pair, or has an acceptable dead right.

$$
\begin{array}{ll}
 & \text{Legal-Name-Instr-Right-Deadok}\,(t,\ n,\ \mathit{instr},\ r) \\
\equiv & \quad \text{Legal-Name-Instr-Right}\,(t,\ n,\ \mathit{instr},\ r) \\
 & \vee\ \Diamond(\quad (\langle \mathit{instr},\ r\rangle = \langle \texttt{'move, 'send}\rangle)[\alpha] \\
 & \qquad \wedge\ \text{dead-right-namep}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond(\quad (\langle \mathit{instr},\ r\rangle = \langle \texttt{'move, 'send-once}\rangle)[\alpha] \\
 & \qquad \wedge\ \text{dead-right-namep}\,(t,\ n)[\alpha]) \\
 & \vee\ \Diamond(\quad (\langle \mathit{instr},\ r\rangle = \langle \texttt{'copy, 'send}\rangle)[\alpha] \\
 & \qquad \wedge\ \text{dead-right-namep}\,(t,\ n)[\alpha])
\end{array}
$$

Port Right Extraction

This section introduces specifications for the various methods of extracting a port right from a target task. The specification Extract-Port-Right is a disjunction of all possibilities.

The following definitions characterize the nine ways a right may be extracted from a task's local name space. These are the six enumerated above and the three ways a dead right may be substituted for a send/send-once right.

Move-Receive differs from Deallocate-Receive-Right in that send rights are not converted to dead rights.

$$\begin{array}{ll}
 & \text{Extract-Port-Right}\,(t,\ n,\ \mathit{instr},\ r,\ p) \\
\equiv & \quad (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'make},\ \texttt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Make-Send}\,(t,\ n,\ p)) \\
 & \vee\ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'copy},\ \texttt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Copy-Send}\,(t,\ n,\ p)) \\
 & \vee\ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'move},\ \texttt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Move-Send}\,(t,\ n,\ p)) \\
 & \vee\ (\quad \Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'move},\ \texttt{'send-once}\rangle)[\alpha] \\
 & \qquad ;\ \Diamond\text{Move-Send-Once}\,(t,\ n,\ p)) \\
 & \vee\ (\quad \Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'make},\ \texttt{'send-once}\rangle)[\alpha] \\
 & \qquad ;\ \Diamond\text{Make-Send-Once}\,(t,\ n,\ p)) \\
 & \vee\ (\quad \Diamond(\langle \mathit{instr},\ r\rangle = \langle \texttt{'move},\ \texttt{'receive}\rangle)[\alpha] \\
 & \qquad ;\ \Diamond\text{Move-Receive}\,(t,\ n,\ p))
\end{array}$$

$$\begin{array}{ll}
 & \text{Move-Receive}\,(t,\ n,\ p) \\
\equiv & \exists\ R \in \textsc{all-rsets},\ \texttt{1} \le i < \textsc{max-refcount}: \\
 & \quad (\quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ R,\ i)[\alpha] \\
 & \qquad ;\quad (\Diamond(R = \{\texttt{'receive}\})[\alpha]\ ;\ \Diamond{\downarrow}\text{local-namep}\,(t,\ n)[\alpha]) \\
 & \qquad\quad \vee\ (\quad \Diamond(R = \{\texttt{'send},\ \texttt{'receive}\})[\alpha] \\
 & \qquad\qquad ;\ \Diamond{\uparrow}\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ i - \texttt{1})[\alpha]))
\end{array}$$

$$\begin{array}{ll}
 & \text{Move-Send}\,(t,\ n,\ p) \\
\equiv & \exists\ R \in \textsc{all-rsets},\ \texttt{1} \le \mathit{refcount} < \textsc{max-refcount}: \\
 & \quad (\quad \Diamond(\text{port-right-rel}\,(t,\ p,\ n,\ R,\ \mathit{refcount})[\alpha] \wedge (\texttt{'send} \in R)[\alpha]) \\
 & \qquad ;\ \Diamond\text{Deallocate-Send-Right}\,(t,\ n,\ \texttt{1}))
\end{array}$$

$$\begin{array}{ll}
 & \text{Move-Send-Once}\,(t,\ n,\ p) \\
\equiv & \quad \Diamond\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send-once}\},\ \texttt{1})[\alpha] \\
 & ;\ \Diamond\text{Remove-Send-Once-Right}\,(t,\ n)
\end{array}$$

Copying and making rights produce no side effect on the source name space. We merely specify pre-conditions on the source that make the extraction possible.

$$\begin{array}{ll}
 & \text{Copy-Send}\,(t,\ n,\ p) \\
\equiv & \exists\ R \in \textsc{all-rsets},\ \texttt{1} \le \mathit{refcount} < \textsc{max-refcount}: \\
 & \quad (\Diamond(\text{port-right-rel}\,(t,\ p,\ n,\ R,\ \mathit{refcount})[\alpha] \wedge (\texttt{'send} \in R)[\alpha]))
\end{array}$$

$$\begin{array}{ll}
 & \text{Make-Send}\,(t,\ n,\ p) \\
\equiv & \exists\ R \in \textsc{all-rsets},\ \texttt{1} \le \mathit{refcount} < \textsc{max-refcount}: \\
 & \quad (\Diamond(\text{port-right-rel}\,(t,\ p,\ n,\ R,\ \mathit{refcount})[\alpha] \wedge (\texttt{'receive} \in R)[\alpha]))
\end{array}$$

$$\begin{aligned}
&\quad \text{Make-Send-Once}\,(t,\ n,\ p)\\
\equiv\ & \exists\ R \in \textsc{all-rsets},\ \mathtt{1} \leq \mathit{refcount} < \textsc{max-refcount}:\\
&\quad (\Diamond(\text{port-right-rel}\,(t,\ p,\ n,\ R,\ \mathit{refcount})[\alpha] \wedge (\mathtt{'receive} \in R)[\alpha]))
\end{aligned}$$

The following are useful composite specifications.

$$\begin{aligned}
&\quad \text{Extract-Send-Right}\,(n,\ \mathit{instr},\ r,\ p)\\
\equiv\ & \quad (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'make},\ \mathtt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Make-Send}\,(\mathtt{t},\ n,\ p))\\
\vee\ & \ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'copy},\ \mathtt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Copy-Send}\,(\mathtt{t},\ n,\ p))\\
\vee\ & \ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'move},\ \mathtt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Move-Send}\,(\mathtt{t},\ n,\ p))\\
\vee\ & \ (\quad \Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'move},\ \mathtt{'send\text{-}once}\rangle)[\alpha]\\
&\quad ;\ \Diamond\text{Move-Send-Once}\,(\mathtt{t},\ n,\ p))\\
\vee\ & \ (\quad \Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'make},\ \mathtt{'send\text{-}once}\rangle)[\alpha]\\
&\quad ;\ \Diamond\text{Make-Send-Once}\,(\mathtt{t},\ n,\ p))
\end{aligned}$$

$$\begin{aligned}
&\quad \text{Extract-Dead-Right}\,(t,\ n,\ \mathit{instr},\ r)\\
\equiv\ & \quad (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'copy},\ \mathtt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Copy-Dead}\,(t,\ n))\\
\vee\ & \ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'move},\ \mathtt{'send}\rangle)[\alpha]\ ;\ \Diamond\text{Move-Dead}\,(t,\ n))\\
\vee\ & \ (\Diamond(\langle \mathit{instr},\ r\rangle = \langle \mathtt{'move},\ \mathtt{'send\text{-}once}\rangle)[\alpha]\ ;\ \Diamond\text{Move-Dead}\,(t,\ n))
\end{aligned}$$

$$\begin{aligned}
&\quad \text{Move-Dead}\,(t,\ n)\\
\equiv\ & \exists\ \mathtt{1} \leq i < \textsc{max-refcount}:\\
&\quad (\Diamond\text{dead-right-rel}\,(t,\ n,\ i)[\alpha]\ ;\ \Diamond\text{Deallocate-Dead-Right}\,(t,\ n,\ \mathtt{1}))
\end{aligned}$$

$$\text{Copy-Dead}\,(t,\ n) \equiv \Diamond\text{dead-right-namep}\,(t,\ n)[\alpha]$$

Port Right Insertion

Here are specifications for the various methods of inserting a port right into a target task. They refer to the more primitive definitions in Section 7.2.1.

We must consider four cases when inserting a send right: either the port is already known by a send right, a receive right, both, or neither. A fifth case can arise as a result of a resource overflow for the port right refcount.

$$\begin{aligned}
&\quad \text{Insert-Port-Right}\,(t,\ p,\ n,\ r)\\
\equiv\ & \quad (\Diamond(r = \mathtt{'send})[\alpha]\ ;\ \Diamond\text{Insert-Send-Right}\,(t,\ p,\ n))\\
\vee\ & \ (\Diamond(r = \mathtt{'receive})[\alpha]\ ;\ \Diamond\text{Insert-Receive-Right}\,(t,\ p,\ n))\\
\vee\ & \ (\Diamond(r = \mathtt{'send\text{-}once})[\alpha]\ ;\ \Diamond\text{Insert-Send-Once-Right}\,(t,\ p,\ n))
\end{aligned}$$

$$\begin{aligned}&\text{Insert-Send-Right}\,(t,\ p,\ n)\\ \equiv\ &\quad \text{New-Send-Right}\,(t,\ p,\ n,\ \mathtt{1})\\ &\vee\ \text{Coalesce-Send-Receive-Right}\,(t,\ p,\ n,\ \mathtt{1})\\ &\vee\ \text{Coalesce-Send-Send-Right}\,(t,\ p,\ n,\ \mathtt{1})\\ &\vee\ \text{Coalesce-Send-Sr-Right}\,(t,\ p,\ n,\ \mathtt{1})\\ &\vee\ \text{Coalesce-Right-Refcount-Pegged}\,(t,\ p,\ n)\end{aligned}$$

A receive right to a port may either be new in this task's name space, or it will coalesce with an existing send right.

$$\begin{aligned}&\text{Insert-Receive-Right}\,(t,\ p,\ n)\\ \equiv\ &\quad \text{New-Receive-Right}\,(t,\ p,\ n)\\ &\vee\ \text{Coalesce-Receive-Send-Right}\,(t,\ p,\ n)\\ &\vee\ \text{Coalesce-Right-Refcount-Pegged}\,(t,\ p,\ n)\end{aligned}$$

$$\begin{aligned}&\text{Coalesce-Right-Refcount-Pegged}\,(t,\ p,\ n)\\ \equiv\ &\exists\ R \in \{\{\mathtt{'send}\},\ \{\mathtt{'send},\ \mathtt{'receive}\}\}:\\ &\quad (\Diamond\text{port-right-rel}\,(t,\ p,\ n,\ R,\ \textsc{max-refcount})[\alpha])\end{aligned}$$

A send-once right does not coalesce with any existing right.

$$\text{Insert-Send-Once-Right}\,(t,\ p,\ n) \equiv \text{New-Send-Once-Right}\,(t,\ p,\ n)$$

7.3 Actions on Virtual Memory

7.3.1 Temporally Allocated

The predicate All-Eventually-Allocated recognizes a computation in which each address has a state in which it is allocated. They are not required to be allocated simultaneously.

Similarly, the predicate All-Eventually-Not-Allocated recognizes a computation in which each address has a state in which it is not allocated.

$$\begin{aligned}&\text{All-Eventually-Allocated}\,(t,\ va,\ l)\\ \equiv\ &\forall\ va \le vpa < (va + l):\ (\Diamond\text{allocated}\,(t,\ vpa)[\alpha])\end{aligned}$$

$$\begin{aligned}&\text{All-Eventually-Not-Allocated}\,(t,\ va,\ l)\\ \equiv\ &\forall\ va \le vpa < (va + l):\ (\Diamond(\neg\ \text{allocated}\,(t,\ vpa))[\alpha])\end{aligned}$$

7.3.2 Extracting and Inserting Data

The predicate Extract-Va-Region recognizes a computation in which the contents of a region of a task's address space are copied into a memory entity. The predicates Insert-In-Line-Data and Insert-Out-Of-Line-Data describe how the contents of a memory entity are copied into a task's address space, either in-line or out-of-line.

```
    Extract-Va-Region $(t,\ va,\ o,\ l,\ m)$
$\equiv \forall\ 0 \le i < l$:
      $(\exists\ (0 \le w < \text{WORDSIZE})$:
         $(\quad \Diamond$va-wordp$(t,\ va + i,\ w)[\alpha]$
           $;\ \ \Diamond\uparrow$m-wordp$(m,\ $trunc-page$(o + i),\ o + i,\ w)[\alpha]))$

    Insert-In-Line-Data $(t,\ va,\ m,\ o,\ l)$
$\equiv \forall\ 0 \le i < l$:
      $(\exists\ (0 \le w < \text{WORDSIZE})$:
         $(\quad \Diamond$m-wordp$(m,\ $trunc-page$(o + i),\ o + i,\ w)[\alpha]$
           $;\ \ \Diamond\uparrow$va-wordp$(t,\ va + i,\ w)[\alpha]))$

    Insert-Out-Of-Line-Data $(t,\ va,\ m,\ o,\ l)$
$\equiv \forall\ 0 \le i < l$:
      $(\quad \Diamond(\neg$ allocated$(t,\ va + i))[\alpha]$
        $;\ \ \Diamond\uparrow$map-rel$(t,\ m,\ $trunc-page$(va + i),\ $trunc-page$(o + i)$,
                    '(read write), '(read write execute), 'copy)$[\alpha])$
```

7.4 Entity Termination

7.4.1 Introduction

When a Mach kernel entity (e.g., task, port) is terminated, a collection of related entities may be terminated along with it. This is because termination of the initial entity may remove the last reference to related entities, which therefore become candidates for garbage collection.

Entity termination is a recursive process: termination of a port may cause termination of other ports. For example, when port p is terminated, all ports for which p is carrying a receive right are also terminated. Side effects may occur as a result of entity termination, as well. When a port is killed, send rights to the port are converted to dead rights.

Termination of an entity is like pulling a burned-out light off of a Christmas tree: all of the burned-out lights along the same string must be pulled off as well. The problem is to know what lights are allowed to remain. This report gives a partial specification for entity termination in the Mach kernel. We completely describe how entity termination recurs. We partially specify the side effects of entity termination.

7.4.2 Informal Rules for Dependent Entity Termination

There are rules that determine, for each entity class, what other entities must be terminated when a member of the class is terminated. The purpose of this section is to state these rules, and to describe side effects of entity termination.

We hypothesize a recursive function $Terminate$ on an entity, that terminates an entity and certain "dependent" entities. The following discussion gives an informal description of entity termination.

Task Termination

When task t is terminated, $Terminate\,(t)$ must additionally perform the following terminations.

- For t's self port p, $Terminate\,(p)$
- For all threads th owned by t, $Terminate\,(th)$.
- For all ports p for which t has the receive right, $Terminate\,(p)$.
- For all memories m which are mapped only into t, $Terminate\,(m)$.

Thread Termination

When thread th is terminated, $Terminate\,(th)$ must additionally perform the following terminations.

- For th's self port p, $Terminate\,(p)$.

Port Termination

When port p is terminated, $Terminate\,(p)$ must additionally perform the following terminations and side effects.

- For all messages mg contained in the port, $Terminate\,(mg)$.
- If p is the object port of some memory m, and m is mapped into no task, $Terminate\,(m)$.
- Side Effect: Convert all send and send-once rights to p to dead rights[3].
- Side Effect: Convert all transit rights to p to dead transit rights.

Message Termination

When a message is received, the message entity is terminated but the contents of the message is copied to the receiver's address space. We are not concerned here with this operation. We are only concerned with the operation that occurs when a message's port is terminated, which requires throwing away the contents of a message. When message mg is terminated, $Terminate\,(mg)$ must additionally perform the following terminations.

- For all ports p for which mg carries a transit receive right, $Terminate\,(p)$.
- For all transit memories m in mg, $Terminate\,(m)$.

Memory Termination

When memory m is terminated, $Terminate\,(m)$ must additionally perform the following terminations.

- For all pages pg representing portions of m, $Terminate\,(pg)$.
- For m's control port p, $Terminate\,(p)$.
- For m's name port p, $Terminate\,(p)$.

[3]Deleting the last send right to a port may result in the enqueuing of a no-more-senders notification. Deleting a send-once rights always results in the enqueing of a send-once notification. We do not model these events.

Page Termination

Terminating a page requires no other entity terminations or side effects.

Processor Termination

Terminating a processor requires no other entity terminations or side effects.

Processor Set Termination

When processor set *pset* is terminated, $Terminate\,(pset)$ must additionally perform the following terminations.

- For *pset*'s self port p, $Terminate\,(p)$.
- For *pset*'s name port p, $Terminate\,(p)$.

Device Termination

Terminating a device requires no other entity terminations or side effects.

7.4.3 A Formal Specification for Entity Termination

We can formally specify entity termination using the temporal logic notation. The following is a set of mutually recursive definitions, each of which captures the informal description of the previous section. The agent of each of the computations is represented by α. In each case, we specify that an entity of the appropriate class is recognized and terminated, and that the appropriate dependent terminations and side effects eventually occur.

```
   $\text{Terminate-Task}\,(t)$
$\equiv$ (   $\Diamond\text{taskp}\,(t)[\alpha]$
   ;      $\Diamond\forall\ p \in \textsc{entities}$:
              $(\text{task-self-rel}\,(t,\ p)[\alpha] \to \text{Terminate-Port}\,(p))$
       $\wedge$  $\Diamond\forall\ th \in \text{threads}\,(t)$:  $\text{Terminate-Thread}\,(th)$
       $\wedge$  $\Diamond\forall\ n \in \mathcal{N},\ p \in \textsc{all-entities}$:
              (   $\text{r-right}\,(t,\ n)[\alpha] \wedge (\text{named-port}\,(t,\ n) = p)[\alpha]$
               $\to \text{Terminate-Port}\,(p))$
       $\wedge$  $\Diamond\forall\ m \in \textsc{entities}$:
              $((\text{mapping-tasks}\,(m) = \{t\})[\alpha] \to \text{Terminate-Memory}\,(m))$
   ;  $\Diamond{\downarrow}\text{entityp}\,(t)[\alpha])$
```

```
   $\text{Terminate-Thread}\,(th)$
$\equiv$ (   $\Diamond\text{threadp}\,(th)[\alpha]$
   ;  $\Diamond\forall\ p \in \textsc{entities}$:
          $(\text{thread-self-rel}\,(th,\ p)[\alpha] \to \text{Terminate-Port}\,(p))$
   ;  $\Diamond{\downarrow}\text{entityp}\,(th)[\alpha])$
```

```
   $\text{Terminate-Port}\,(p)$
$\equiv$ (   $\Diamond\text{portp}\,(p)[\alpha]$
   ;      $\Diamond\forall\ t \in \textsc{all-entities},\ n \in nset,\ 0 \le i < \textsc{max-refcount}$:
              (   $\text{port-right-rel}\,(t,\ p,\ n,\ \{\texttt{'send}\},\ i)[\alpha]$
               $\to \Diamond{\uparrow}\text{dead-right-rel}\,(t,\ n,\ i)[\alpha])$
       $\wedge$  $\Diamond\forall\ mg \in \textsc{all-entities},\ r \in \mathcal{R},\ 0 \le i < \textsc{max-msg-size}$:
              (   $\text{transit-right-rel}\,(mq,\ p,\ r,\ i)[\alpha]$
               $\to \Diamond{\uparrow}\text{null-message-element-rel}\,(mg,\ \texttt{'dead-right},\ i)[\alpha])$
       $\wedge$  $\Diamond\forall\ mg \in \textsc{all-entities}$:
              $((mg \in \text{messages}\,(p))[\alpha] \to \text{Terminate-Message}\,(mg))$
       $\wedge$  $\Diamond\forall\ m \in \textsc{all-entities}$:
              (   $\text{object-port-rel}\,(m,\ p) \wedge \neg\ \text{mapped}\,(m)$
               $\to \text{Terminate-Memory}\,(m))$
   ;  $\Diamond{\downarrow}\text{entityp}\,(p)[\alpha])$
```

$$
\begin{array}{l}
\quad \text{Terminate-Message}\,(mg) \\
\equiv (\quad \Diamond\text{messagep}\,(mg)[\alpha] \\
\quad ;\quad \Diamond\forall\ p \in \text{ALL-ENTITIES},\ 0 \le i < \text{MAX-MSG-SIZE}: \\
\qquad\qquad (\quad \text{transit-right-rel}\,(mg,\ p,\ \texttt{'receive},\ i)[\alpha] \\
\qquad\qquad\ \ \to \text{Terminate-Port}\,(p)) \\
\qquad \wedge\ \Diamond\forall\ m \in \text{ALL-ENTITIES},\ 0 \le i < \text{MAX-MSG-SIZE}: \\
\qquad\qquad (\qquad \text{exists-transit-memory}\,(mg,\ i)[\alpha] \\
\qquad\qquad\quad \wedge\ (\text{trans-memory}\,(mg,\ i) = m)[\alpha] \\
\qquad\qquad\ \ \to \text{Terminate-Memory}\,(m)) \\
\quad ;\ \Diamond{\downarrow}\text{entityp}\,(mg)[\alpha])
\end{array}
$$

$$
\begin{array}{l}
\quad \text{Terminate-Memory}\,(m) \\
\equiv (\quad \Diamond\text{memoryp}\,(m)[\alpha] \\
\quad ;\quad \Diamond\forall\ p \in \text{ENTITIES}: \\
\qquad\qquad (\text{control-port-rel}\,(m,\ p)[\alpha] \to \text{Terminate-Port}\,(p)) \\
\qquad \wedge\ \Diamond\forall\ p \in \text{ENTITIES}: \\
\qquad\qquad (\text{name-port-rel}\,(m,\ p)[\alpha] \to \text{Terminate-Port}\,(p)) \\
\qquad \wedge\ \Diamond\forall\ pg \in \text{ENTITIES},\ 0 \le o < \text{MEMORYSIZE}: \\
\qquad\qquad (\text{represents-rel}\,(pg,\ m,\ o) \to \text{Terminate-Page}\,(pg)) \\
\quad ;\ \Diamond{\downarrow}\text{entityp}\,(m)[\alpha])
\end{array}
$$

$$\text{Terminate-Page}\,(pg) \equiv \Diamond\text{pagep}\,(pg)[\alpha]\ ;\ \Diamond{\downarrow}\text{entityp}\,(pg)[\alpha]$$

$$\text{Terminate-Proc}\,(proc) \equiv \Diamond\text{procp}\,(proc)[\alpha]\ ;\ \Diamond{\downarrow}\text{entityp}\,(proc)[\alpha]$$

$$
\begin{array}{l}
\quad \text{Terminate-Procset}\,(procset) \\
\equiv (\quad \Diamond\text{procsetp}\,(procset)[\alpha] \\
\quad ;\quad \Diamond\forall\ p \in \text{ENTITIES}: \\
\qquad\qquad (\text{procset-self-rel}\,(procset,\ p)[\alpha] \to \text{Terminate-Port}\,(p)) \\
\qquad \wedge\ \Diamond\forall\ p \in \text{ENTITIES}: \\
\qquad\qquad (\quad \text{procset-name-port-rel}\,(procset,\ p)[\alpha] \\
\qquad\qquad\ \ \to \text{Terminate-Port}\,(p)) \\
\quad ;\ \Diamond{\downarrow}\text{entityp}\,(procset)[\alpha])
\end{array}
$$

$$\text{Terminate-Device}\,(d) \equiv \Diamond\text{devicep}\,(d)[\alpha]\ ;\ \Diamond{\downarrow}\text{entityp}\,(d)[\alpha]$$

We can observe several instances of mutual recursion in the above definitions. For example, we have the following.

Terminate-Port $(p) \longrightarrow$ Terminate-Message $(mg) \longrightarrow$ Terminate-Port (p)

This chain represents the following sequence:

- Terminate-Port (p)
- For all messages mg contained in the port, Terminate-Message (mg).
- For all ports p for which mg carries a transit receive right, Terminate-Port (p).

Can we be sure that such a recursion completes? If port termination runs without interference, e.g., in the absence of concurrent kernel computations, then there is a danger that the recursion does not complete only if there can be a cycle in the state graph that involves ports and transit receive rights. In [BS94b] we specify that in a legal kernel state there are no such cycles. For example, port p_1 may not contain a message that carries a receive right for port p_2 if p_2 contains a message that carries a receive right for port p_1. Without this requirement on a legal state, an implementation of port termination would have to check for cycles in the state graph to avoid infinite recursion.

In the presence of concurrency in a kernel computation, it is possible for two processes to collaborate to extend this recursion indefinitely, or at least until memory resources are exhausted. Suppose that process A terminates port p_1. In parallel, process B executes a loop in which it successively creates port p_i and sends the receive right of port p_{i-1} to p_i. Process A's call to Terminate-Port may be interleaved with process B in such a way as to cause Terminate-Port to follow an arbitrarily long sequence of recursive calls.

The above scenario may be extremly unlikely. However, it serves to illustrate the point that the length of an entity termination computation may depend on concurrent activity. On a system with inexhaustible resources, it is conceivable that entity termination may not complete.

Bibliography

[BS94a] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., December 1994.

[BS94b] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Entities and relations. Technical Report 88, Computational Logic, Inc., December 1994.

[Loe91] Keith Loepere. Mach 3 kernel interface. Technical report, Open Software Foundation, May 1991.

Index