Technical Note

DTOS FORMAL SECURITY POLICY MODEL (FSPM) (Non-Z Version)

Secure Computing Corporation

Abstract

This report identifies the services provided by the DTOS kernel and the security requirements governing when the kernel provides the services.

Part Number	83-0902023A001
Created	
Revised	26 September 1996
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/home/cmt/rev/dtos/docs/fspm/RCS/fspm-driver.vdd,v 1.25 26 September 1996`

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

Contents

Section 1 Scope

1.1 Identification

This Formal Security Policy Model (FSPM) states the security policy for the prototype kernel developed on the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209.

1.2 System Overview

The DTOS prototype is an enhanced version of the CMU Mach 3.0 kernel that provides support for a wide variety of security policies by enforcing access decisions provided to it by a *security server*. The set of policies that can be supported is determined by the *control points* implemented in the prototype. Logically, an access computation query is sent to a security server whenever the DTOS kernel reaches a control point. Request processing cannot continue until the security server informs the DTOS kernel whether the security policy allows the processing to be performed. For efficiency reasons, the DTOS kernel is permitted to cache security decisions made by security servers. Ideally, most access checks can be performed by looking up entries in a cache rather than actually querying a security server.

By implementing different security servers, a wide range of policies can be supported by the same DTOS kernel. By implementing a security server that allows all accesses, the DTOS kernel behaves essentially the same as the CMU Mach 3.0 kernel. Although this is uninteresting from a security standpoint, it demonstrates the compatibility of DTOS with Mach 3.0. By using appropriately developed security servers, the DTOS kernel can support interesting security policies such as MLS (multi-level security) and type enforcement.

1.3 Document Overview

The report is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Applicable Documents**, describes other documents that are relevant to this document.
- Section 3, **FSPM Overview** provides motivation for the DTOS approach to security and an overview of the approach used to present the policy.
- Section 4, **Basic Kernel State Definition**, provides a brief description of the Mach 3.0 kernel data structures.
- Section 5, **DTOS State Extensions**, describes new kernel data structures required by the DTOS design.
- Section 6, **DTOS Services**, describes the services provided by the DTOS kernel.
- Section 7, **Base Kernel Policy**, describes the security requirements governing the DTOS kernel's enforcement of a policy specified by a security server.
- Section 8, **Generic Security Server Requirements**, provides a general framework for DTOS security servers.

- Section 9, **Notes**, contains a list of acronyms, a glossary, and a description of open issues relevant to the FSPM.
- Appendix A, **Bibliography**, provides the bibliographical information for the documents referenced in the FSPM.
- Appendix B, **Prototype Security Server Requirements**, describes the rules the prototype security server uses for computing accesses.

Note that while this report contains a description of the DTOS system state and services, it makes no attempt at providing a complete description. Readers that are unfamiliar with Mach and/or DTOS should consult references [3], [4], and [10].

Readers who are interested in a more formal presentation of this document should consult a separate version of this report (reference [8]) that provides a Z formalization of the definitions and requirements stated here.

Section 2 Applicable Documents

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

- OSF Mach Kernel Principles [4]

Although Section 4 provides a summary of the information contained in reference [4], some readers might need to consult the more complete information in reference [4].

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

- OSF Mach 3 Kernel Interface [3]
- DTOS Kernel Interface Document (KID) [10]
- DTOS Kernel and Security Server Software Design Document (SDD) [7]

Although an understanding of these documents is desirable, such an understanding is not necessary to understand the majority of this document.

The following documents provide formal descriptions of certain aspects of Mach-like systems:

- CLI Mathematical Model of Mach [1]
- DTMach FTLS [2]
- DTOS FTLS [6]

The model of Mach described in Section 4 is derived from references [1] and [2]. Although the material presented in Section 4 is intended to be self contained, some readers might want to consult references [1] and [2] for more details. Reference [2] is an earlier version of reference [6]. The former provides more complete coverage of the requests, while the latter provides more readable and more correct descriptions.

The following document is the standard reference for DoD security policies:

- DoD Trusted Computer System Evaluation Criteria [5]

Some of the motivational examples in this document assume a basic understanding of security policies as defined in reference [5].

Readers who are interested in a Z formalization of the model presented in this document should consult the following reference:

- DTOS Formal Security Policy Model [8]

Section 3 FSPM Overview

This section provides an overview of the DTOS FSPM. In addition to providing a high level description, this section also highlights differences between the DTOS FSPM and typical FSPMs.

3.1 Policy Development Approach

The goal of a security policy is to protect the confidentiality, integrity, and availability of the system against attacks by malicious users and mistakes made by innocent users. For example, the goal of an MLS policy[5] is to prevent users from obtaining information for which they are not cleared. As another example, a system policy that prohibits users other than system administrators from rebooting the machine addresses denial of service as the result of a mistake made by a user.

Traditionally, there have been two related but distinct approaches to developing security policies. The first approach, the *threat based* approach is to identify the system threats that are of concern and develop requirements that address the threats. The second approach, the*criteria based* approach is to interpret a set of requirements specified by an evaluation criteria document (such as [5]) for the target system. The relation between the two approaches is that in the second approach it is assumed that the developers of the evaluation criteria have already identified all of the relevant threats.

The criteria based approach is infeasible for DTOS due to the goal to support a wide range of policies. Regardless of whether an evaluation criteria document contains MLS, integrity, or availability requirements, there is always the possibility that the user of a DTOS system will want to enforce some other type of security. Consequently, the DTOS policy must provide a framework in which a variety of policies can be supported rather than simply interpreting requirements in an existing evaluation criteria.

Thus, the DTOS policy development is threat based. However, the threats identified are of a different nature than those traditionally identified. When developing the policy for a system that is intended to enforce a single policy, the identified threats typically are specific to that policy. For example, while covert channels[5] are a threat with respect to MLS policies, they are typically not a threat with respect to integrity policies. Since the DTOS policy is intended to provide a framework that supports a wide variety of policies, the threats identified for DTOS must be policy independent.

The intent is for users to be able to counter threats to their systems by appropriately configuring DTOS. Furthermore, as the set of threats against which a site must protect evolves, administrators should be able to reconfigure DTOS to address the new set of threats. This requires controls to be placed on essentially all services. For example, DTOS must control the setting of the scheduling policy for a thread since some users will want to protect against service denial to user threads. Although the denial of service threat might be of little concern to most users, the possibility that some users might be concerned suggests viewing it as a real threat. Since providing protection against every conceivable threat is impossible, a judgement call has been made on the set of threats that are of concern.

The approach taken in the remainder of this document is to view any access of the kernel

state as being a potential threat. By viewing each access as a potential threat and providing appropriate control mechanisms, the goal of supporting multiple policies can be achieved.[1] In Section 6, we characterize the various types of accesses that can be made to the DTOS state as DTOS "services." For motivational purposes, we also provide some specific examples in Section 6 of how the threats associated with DTOS services relate to more general threats to computing systems.

In stating the policy, we categorize each access as being either an *abstract service* or an *implementation service*. An abstract service is characterized by a relation on pairs of system states that specifies a change to a kernel data structure. For example, the service that creates a new task is characterized by a relation that specifies that the new system state contains a task that was not present in the old system state. Stating a policy on when an abstract service can be provided allows modifications to the kernel data structures associated with the service to be controlled.

Note that the same abstract service can be provided by multiple requests. For example, **mach_port_allocate** and **mach_port_allocate_name** both provide the abstract service of adding a name to a task's name space.[10]

An implementation service is a specific Mach request. When it is difficult to formally define the abstract services associated with a specific Mach request, we address the request by controlling when the request itself can be invoked rather than controlling the abstract services provided by the request. In some cases, the reason for being unable to identify the abstract services is that the request alters data structures that are not visible at the level of the policy. For example, the **host_adjust_time** request (see reference [10]) alters the rate at which the system clock is updated, while the model presented in this document does not address the kernel data structures controlling the rate at which the system clock is updated. In these cases, the implementation services could be replaced by abstract services by developing a more detailed system model.

The other primary examples of implementation services are services that "observe" rather than merely "modify" the system state. Observations are more difficult to detect than modifications since they do not leave a trace in the system state. A modification of a state component results in its value changing, while an observation does not.

Regardless of whether a service is an abstract service or an implementation service, we associate a *permission* with the service. Note that the majority of this document is devoted to defining the abstract and implementation services provided by the kernel and the permissions enforced by the kernel. Once these are defined, the policy is essentially:

> *The kernel ensures that each of the defined services is provided only when the client of the service has the appropriate permission.*

This policy controls only the providing of the defined services. Although the services defined in this document are intended to define all of the interesting services provided by the DTOS microkernel, we might have failed to identify some of the provided services. The policy places no constraints on the providing of any such services.

To implement the policy, the set of services provided by each request must be identified. Once these services are identified, the requirements in this document can be used to derive the set of permission checks that must be performed. The DTOS Kernel Interface Document [10] and the DTOS Kernel and Security Server Software Design Document [7] identify a set of permission checks that are currently performed for each request. We are in the process of examining

[1] See the DTOS Generalized Security Policy Specification [9] for more on supporting multiple policies.

the consistency of these permission checks with those called for by the requirements in this document.

3.2 Separation of Enforcer and Decider

Another way in which the DTOS FSPM differs from a traditional security policy is that the requirements on the enforcer of the policy are separate from the requirements on the definer of the policy. Traditionally, the distinction between enforcer and decider has been abstracted away. For example, the *-Property (see [5]):

> *A subject may only write objects at its level or above.*

directly binds the abstract service of writing to an object with the MLS requirement. In DTOS, the set of permissions provides an intermediary between the enforcer and the decider. The kernel associates a permission with an abstract service and each security server associates a security requirement with the permission. By enforcing the access computations that a security server communicates as allowed permissions, the kernel can properly enforce the policy defined by the security server.

In addition to supporting policy flexibility, explicitly addressing the separation of the kernel and security server functionality provides a cleaner mapping of the policy to the implementation. The requirements stated in Section 7 provide guidance for the implementation of the DTOS kernel, while the requirements stated in Section 8 and Appendix B state requirements that provide guidance for the implementation of the prototype security server. The mapping from implementation to policy is so direct that *the tables in Section 7 that define the DTOS formal security policy are automatically generated from a file that is also used to generate portions of the DTOS SDD[7] and kernel code. As opposed to manual updating of all of these components, this automated approach provides much greater confidence that the implementation of the DTOS kernel actually satisfies the policy.* [2]

Editorial Note:
This document currently defines only those permissions and services relevant to the microkernel. To address a user space server such as a file server, definitions would need to be given for each of the services to be controlled and permissions would have to be defined to control each service. In addition, if the server policies were layered on top of the microkernel policy, discussion would need to be added concerning how the server and microkernel policies were related. Once policies were stated for each of the servers comprising the trusted operating system, an overall system policy could be stated. Although this is a desirable end result, the current scope of the DTOS program is only the microkernel.

To unambiguously define the abstract services provided by DTOS, we must first provide a precise definition of the DTOS data structures. Sections 4 and 5 provide such a description by describing an abstract model of the DTOS kernel. Section 6 contains a description of the DTOS services in the context of this abstract model. The remaining sections use the abstract model and service definitions to state the policy.

[2] Of course, this approach still requires coordination between any individual who changes the security policy file and those individuals responsible for regenerating documents from the file.

Section 4

Basic Kernel State Definition

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

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

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

As a convention, we underline the first letter in the identifier for each primitive structure in the Mach state. This is most useful when identifying which primitive structures are affected by the DTOS services defined in Section 6.

4.1 Primitive Entities

The primitive entities in Mach are:

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

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

Ports — unidirectional communication channels between tasks

Messages — entities transmitted through ports

Memories — memory object representing a shared memory

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

Hosts — instances of the Mach kernel

Processors — devices capable of executing threads

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

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

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

At any given time, only certain primitive entities are present in the system. The sets $\underline{task\_exists}$, $\underline{thread\_exists}$, $\underline{port\_exists}$, $\underline{message\_exists}$, $\underline{memory\_exists}$, $\underline{page\_exists}$, $\underline{proc\_exists}$, $\underline{procset\_exists}$, and $\underline{device\_exists}$ denote the entities of each class that are present in the current system state.

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

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

4.2 Process Management

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

4.2.1 Thread to Task Relationship

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

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

4.2.2 Execution Status

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

The values of this type have the following meanings:

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

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

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

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

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

4.2.3 Priority Levels

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

than a second priority. The constants $Lowest\_possible\_priority$ and $Highest\_possible\_priority$ denote the maximum and minimum integers, respectively, in $Priority\_levels$.

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

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

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

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

The priority level of a thread can temporarily be depressed by the request **swtch_pri** or **thread_switch** to allow other threads to run. When a thread is depressed, its priority is set to the lowest possible priority.[3] The set $\underline{d}epressed\_threads$ denotes those threads whose priority is currently depressed. The expression $\underline{p}riority\_before\_depression(thread)$ denotes the priority level $thread$ had before depression if $thread$'s priority level has been depressed and $\underline{t}hread\_priority(thread)$ otherwise.

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

4.2.4 Scheduling Policies

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

4.2.5 Instruction Pointer

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

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

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

4.2.6 Emulation Environment

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

4.2.7 Sampling

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

There are six recognized sample types. They are:

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

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

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

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

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

The same sampling information is kept for tasks.

4.2.8 Thread Time Statistics

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

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

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

4.2.9 Machine State

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

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

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

4.3 Port Name Space

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

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

4.3.1 Port Rights

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

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

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

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

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

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

For convenience:

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

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

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

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

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

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

4.3.2 Port Sets

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

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

4.3.3 Dead Rights

A dead name is a name which previously named a send, receive, or send-once right for a task, but no longer does.[4] Each dead name in a task can have an associated count that is analogous to the reference count associated with send rights. This count is initially set based on the user reference counts for the right previously bearing the name. The count may be modified by subsequent actions of the kernel. The relation $\underline{dead\_right\_rel}$ identifies the dead names and their associated counts for each task; an element $(task, name, i)$ is an element of $\underline{dead\_right\_rel}$ if $name$ is a dead name in $task$ with associated count i. The previously defined constant, Max_right_refs, is a system-wide maximum for the reference count of a given dead right. For convenience:

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

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

4.3.4 Summary

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

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

4.4 Ports

This section describes data structures associated with ports.

4.4.1 Make Send Count

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

4.4.2 Message Queues

Each port has an associated message queue. A message queue can be thought of as a sequence of messages. In Mach, a task may set a limit on the number of messages that are permitted in a given message queue. The value $Mach\_port\_q\_limit\_default$ represents the default limit

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

the kernel uses for newly allocated ports. The value $Mach\_port\_q\_limit\_max$ represents a system-imposed limit on the value a task may specify as the limit for a message queue.

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

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

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

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

4.5 Notifications

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

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

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

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

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

The relation $\underline{port\_notify\_dead\_rel}$ identifies the task-name pairs for which a dead-name notification has been requested and the associated notification ports. For convenience, $port\_notify\_dead(task, name)$ is used to denote the notification port registered for a dead-name notification on $name$ in $task$'s name space.

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

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

4.6 Special Ports

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

4.6.1 Task Ports

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

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

For convenience:

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

4.6.2 Thread Ports

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

4.6.3 Memory Ports

A kernel and a memory object interact by engaging in a dialogue. The kernel sends messages to an object port and the object manager sends messages to a control port. There is also a

name port used to identify the object in **vm_region** requests. The relations $\underline{object\_port\_rel}$, $\underline{control\_port\_rel}$, and $\underline{name\_port\_rel}$ are used to represent the binding between a memory and its associated ports. For a particular Mach host kernel, there is at most one of each type of port associated with a given memory. Furthermore, no object port is associated with more than one memory object. For convenience:

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

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

4.6.4 Host Ports

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

4.6.5 Processor Ports

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

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

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

4.6.6 Device Ports

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

- The expression $device\_port(dev)$ is used to denote dev's device port.

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

4.6.7 Device Master Port

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

4.6.8 Summary

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

Editorial Note:
The following needs some revision:

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

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

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

4.7 Total Send Rights

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

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

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

4.8 Registered Rights

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

4.9 Memory System

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

4.9.1 Memory

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

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

- $Memory\_copy\_none$ —

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

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

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

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

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

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

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

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

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

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

Review Note:
Need to figure out how $\underline{default\_mem\_manager}$ relates to $\underline{managed}$ and $\underline{manager}$.

4.9.2 Pages

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

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

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

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

Mach allows pages to be locked against particular types of accesses. This is represented by associating a set of protections with each page. The protections are of type $PROTECTION$ which

is comprised of the elements $Read$, $Write$, and $Execute$. The relation $page\_lock\_rel$ indicates the access modes against which a page is locked. For convenience $page\_locks(page)$ is defined to be the set of access modes against which $page$ is locked.

4.9.3 Address Space

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

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

4.9.4 Memory Protection

Mach protects memory objects by assigning protections to each page in a task's address space. Three sets of protections are associated with each page in a task's address space. The Mach protection holds currently applicable protection limits as indicated by users. The maximum protection limits the allowable values for the Mach protection. The third set, the current protections, is what actually limits a task's access to a page. This is a DTOS addition and will be further defined in Section 5.9.[5]

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

4.9.5 Memory Inheritance

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

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

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

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

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

4.9.6 Shadow Memories

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

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

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

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

4.9.7 Page Wiring

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

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

4.10 Messages

This section discusses the structure of messages.

4.10.1 Message Options

The type *MACH_MSG_OPTION* denotes the base values of the *options* parameter of **mach_msg**. The recognized values of this type are *Mach_send_msg*, *Mach_rcv_msg*, *Mach_send_cancel*, *Mach_send_notify*, *Mach_rcv_notify*, *Mach_rcv_large*, *Mach_send_timeout*, and *Mach_rcv_timeout*. The *options* parameter is set to some set of the base values.

4.10.2 Complex Messages

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

4.10.3 Data Types

Each element in the body of a message is typed. The set *MACH_MSG_TYPE* denotes the set of data types recognized by the system.

Whenever a port right is sent in a message, the client indicates a transfer option for the port right. The collection of acceptable transfer options is denoted by *Recognized_transfer_options* and contain the values *Mmt_make_send*, *Mmt_copy_send*, *Mmt_move_send*, *Mmt_make_send_once*, *Mmt_move_send_once*, and *Mmt_move_receive*.

An element of type *Mmt_make_send* indicates a receive right held by the sender from which a send right is to be created for the receiver. Similarly, an element of type *Mmt_make_send_once* indicates a receive right held by the sender from which a send-once right is to be created for the receiver.

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

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

After the kernel translates the port rights to an internal representation, it is no longer relevant whether the right was moved, copied or made and the kernel simply records the type of right, *Mach_msg_type_port_receive*, *Mach_msg_type_port_send*, or *Mach_msg_type_port_send_once*. These values of *MACH_MSG_TYPE* comprise the set *Mach_msg_type_port_rights*.

4.10.4 Message Headers

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

- *local_port* — specifies the reply port when sending a message (*Mach_port_null* indicates no reply port is specified)
- *local_rights* — the port rights for the local port (if one is specified)
- *remote_port* — specifies the destination port when sending a message
- *remote_rights* — the port rights for the remote port
- *size* — specifies the size, in bytes, of a message when receiving
- *msg_sequence_no* — specifies the sequence number when receiving a message
- *operation* — operation or function id set by message sender

In addition, a message header in kernel space contains a value *complex* which indicates whether the message carries port rights or memory regions or both. This value is a set of elements of type *COMPLEX_OPTION*. In place of *complex*, a message header in user space contains a single value *complex_boolean* indicating whether the message carries port rights and/or memory regions. The possible values are *Co_carries_rights_and_or_memory* and *Co_carries_neither_rights_nor_memory*. If *complex_boolean* has value *Co_carries_neither_rights_nor_memory*, then the message contains no port rights nor memory regions regardless of what is indicated by the individual data elements of the message.

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

4.10.5 Outcall Operations

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

- *Exception_ids* to denote the set of operations used by the kernel when sending an exception message, The only element of this set is *Mach_exception_id*.
- *Kernel_service_reply_ids* to denote the set of operations used by the kernel in reply messages to kernel service requests,
- *Security_server_ids* to denote the set of security server operations,
- *Audit_ids* to denote the set of audit operations,
- *Mem_obj_confirmation_ids* to denote the set of operations used by the kernel when sending confirmations of memory operations to a pager,
- *Pager_request_ids* to denote the set of pager operations,
- *Mach_notify_ids* to denote the set of operations used by the kernel in notification messages, and
- *Network_packet_ids* to denote the set of operations used by the kernel when forwarding network packets.

4.10.6 Message Bodies

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

- the number of data elements contained in the message element

- a data type
- a collection of data elements or a single address

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

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

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

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

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

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

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

- $Msg\_value(n, mach\_msg\_type, (task, value\_seq))$ — an in-line element; if $mach\_msg\_type$ is an element of $Recognized\_transfer\_options$ and some elements of $value\_seq$ have not yet been resolved to ports then further processing is required to transform the sequence of data into a sequence of ports.
  Note that there are two forms for elements of $value\_seq$. An entry of the form $V\_data(msg\_data, v\_data\_l)$ denotes the data msg_data while an entry of the form $V\_port(port, v\_data\_l)$ denotes a port name that has been resolved into a port. In either case, v_data_l indicates whether the element came from an in-line data element or an out-of-line data element. The only time v_data_l will indicate an out-of-line data element is when the element is a port name from an out-of-line data element that has been resolved into a port.
- $Transit\_right(n, mach\_msg\_type, (task, port\_seq, v\_data\_l))$ — a sequence of port rights in transit; $task$ indicates the task that sent the message and v_data_l indicates whether the port right was sent in-line or out-of-line
- $Msg\_region(n, mach\_msg\_type, (task, va, olsd))$ — an out-of-line element that requires further processing to transform the task-address pair into a memory-offset pair; $task$ indicates the task that sent the message and $olsd$ indicates whether the region should be deallocated from $task$'s address space
- $Transit\_memory(n, mach\_msg\_type, (task, memory, offset))$ — an out-of-line element that has been transformed from a task-address pair to a memory-offset pair; $task$ indicates the task that sent the message

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

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

4.10.7 Message Status

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

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

These elements comprise the values of the type MSG_STATUS.

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

4.10.8 Message Structure

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

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

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

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

4.10.9 Pending Receives

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

- $notify$ — the notify port name specified by the receiving task
- $option$ — the options specified by the receiving task
- rcv_size — the receive size specified by the receiving task

- $time\_out\_at$ — the time at which the request will time out; this has the same format as the $time\_out\_at$ component of $InternalMessage$.

4.10.10 Reply Ports

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

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

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

4.10.11 Summary

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

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

4.11 Processors and Processor Sets

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

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

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

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

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

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

4.12 Time

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

4.13 Devices

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

A kernel-space device driver may supply event counters for use by user-space device drivers. An event counter is used as a semaphore for events produced by kernel-space drivers. The counter is incremented when a relevant event occurs and decremented when a thread (e.g., a user-space device driver) indicates via the **evc_wait** trap that it wishes to process an event. Each task refers to an event by referencing its event counter. The appropriate event counter is communicated to a thread in a driver-specific way.[6] The expression $EVENT\_COUNTER$ denotes the set of all event counters.

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

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

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

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

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

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

4.14 Summary

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

Section 5

DTOS State Extensions

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

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

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

The organization of this section is as follows:

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

5.1 Subject Security Information

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

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

The expressions $\underline{t}ask\_sid(task)$, $task\_mid(task)$ and $task\_aid(task)$ are used to denote the SSI,

MID and AID associated with a task. The expression $thread\_sid(thread)$ denotes the SSI associated with a thread. It is defined to be the SSI of its parent task.

5.2 Object Security Information

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

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

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

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

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

5.3 Security Identifiers for Access Computations

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

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

One of the features of Mach is that it allows tasks to perform operations on other tasks that have not traditionally been provided by operating systems. For example, Mach allows tasks to access memory regions in other tasks while one of the features of traditional operating systems is the separation of address spaces. To provide finer control over task accesses, we define $Task\_self\_sid$ to be a value to be used in access computations governing accesses a task makes to itself. Similarly, we use $Thread\_self\_sid$ to be a value to be used in access computations governing accesses a task makes to threads that it owns. The security policy should normally be defined in such a way as to prevent any kernel entities from being assigned $Task\_self\_sid$ or

$Thread\_self\_sid$ as their SID.[7] Instead, these SIDs indicate to security servers that the kernel requires an access computation to be performed between a task and the task itself or between a task and one of the task's threads. One potential use of this finer control would be to contain a faulty task by preventing it from corrupting other tasks having the same SID.

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

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

5.4 Permissions

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

- associating a set of allowed permissions[8] with each SSI-OSI pair,
- associating a set of required permissions with each service, and
- granting service only when the required permissions are contained in the allowed permissions for the client to the target for the operation.

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

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

The elements of $Kernel\_permission$ are enumerated in subsections 5.4.1-5.4.14.

5.4.1 IPC Permissions

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

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

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

5.4.2 Port Permissions

The DTOS kernel enforces the following permissions on port requests: *Add_name*, *Alter_pns_info*, *Extract_right*, *Lookup_ports*, *Manipulate_port_set*, *Observe_pns_info*, *Port_rename*, *Register_notification*, *Register_ports*, *Remove_name*. We use *Port_permissions* to denote this set of permissions.

5.4.3 VM Permissions

The DTOS kernel enforces the following permissions on VM requests: *Access_machine_attribute*, *Allocate_vm_region*, *Chg_vm_region_prot*, *Copy_vm*, *Deallocate_vm_region*, *Get_vm_region_info*, *Get_vm_statistics*, *Read_vm_region*, *Set_vm_region_inherit*, *Wire_vm_for_task*, *Write_vm_region*. We use *Vm_permissions* to denote this set of permissions.

5.4.4 Memory Object Permissions

The DTOS kernel enforces the following permissions on memory requests: *Have_execute*, *Have_read*, *Have_write*, *Page_vm_region*. We use *Memory_object_permissions* to denote this set of permissions.

5.4.5 Pager Permissions

The DTOS kernel enforces the following permissions on pager requests: *Change_page_locks*, *Destroy_object*, *Get_attributes*, *Invoke_lock_request*, *Make_page_precious*, *Provide_data*, *Remove_page*, *Revoke_ibac*, *Save_page*, *Set_attributes*, *Set_ibac_port*, *Supply_ibac*. We use *Pager_permissions* to denote this set of permissions.

5.4.6 Thread Permissions

The DTOS kernel enforces the following permissions on thread requests: *Abort_thread*, *Abort_thread_depress*, *Assign_thread_to_pset*, *Can_swtch*, *Can_swtch_pri*, *Depress_pri*, *Get_thread_assignment*, *Get_thread_exception_port*, *Get_thread_info*, *Get_thread_kernel_port*, *Get_thread_state*, *Initiate_secure*, *Raise_exception*, *Resume_thread*, *Sample_thread*, *Set_max_thread_priority*, *Set_thread_exception_port*, *Set_thread_kernel_port*, *Set_thread_policy*, *Set_thread_priority*, *Set_thread_state*, *Suspend_thread*, *Switch_thread*, *Terminate_thread*, *Wait_evc*, *Wire_thread_into_memory*. We use *Thread_permissions* to denote this set of permissions.

5.4.7 Task Permissions

The DTOS kernel enforces the following permissions on task requests: *Add_thread*, *Add_thread_secure*, *Assign_task_to_pset*, *Change_sid*, *Chg_task_priority*, *Create_task*, *Create_task_secure*, *Cross_context_create*, *Cross_context_inherit*, *Get_emulation*, *Get_task_assignment*, *Get_task_boot_port*, *Get_task_exception_port*, *Get_task_info*, *Get_task_kernel_port*, *Get_task_threads*, *Make_sid*, *Resume_task*, *Sample_task*, *Set_emulation*, *Set_ras*, *Set_task_boot_port*, *Set_task_exception_port*, *Set_task_kernel_port*, *Suspend_task*, *Terminate_task*, *Transition_sid*. We use *Task_task_permissions* to denote this set of permissions.

We use *Task_permissions* to denote the union of *Task_task_permissions*, *Port_permissions*, and *Vm_permissions*.

5.4.8 Host Name Port Permissions

The DTOS kernel enforces the following permissions on host name port requests: *Create_pset*, *Flush_permission*, *Get_audit_port*, *Get_authentication_port*, *Get_crypto_port*, *Get_default_pset_name*, *Get_host_control_port*, *Get_host_info*, *Get_host_name*, *Get_host_version*, *Get_negotiation_port*, *Get_network_ss_port*, *Get_security_master_port*, *Get_security_client_port*, *Get_special_port*, *Get_time*, *Pset_names*, *Set_audit_port*, *Set_authentication_port*, *Set_crypto_port*, *Set_negotiation_port*, *Set_network_ss_port*, *Set_security_master_port*, *Set_security_client_port*, *Set_special_port*. We use *Host_name_port_permissions* to denote this set of permissions.

5.4.9 Host Control Port Permissions

The DTOS kernel enforces the following permissions on host control port requests: *Get_boot_info*, *Get_host_processors*, *Pset_ctrl_port*, *Reboot_host*, *Set_default_memory_mgr*, *Set_time*, *Wire_thread*, *Wire_vm*. We use *Host_control_port_permissions* to denote this set of permissions.

5.4.10 Processor Permissions

The DTOS kernel enforces the following permissions on processor requests: *Assign_processor_to_set*, *Get_processor_assignment*, *Get_processor_info*, *May_control_processor*. We use *Processor_permissions* to denote this set of permissions.

5.4.11 Processor Set Name Port Permissions

The DTOS kernel enforces the following permissions on processor set name port requests: *Get_pset_info*. We use *Procset_name_port_permissions* to denote this set of permissions.

5.4.12 Processor Set Control Port Permissions

The DTOS kernel enforces the following permissions on processor set control port requests: *Assign_processor*, *Assign_task*, *Assign_thread*, *Chg_pset_max_pri*, *Define_new_scheduling_policy*, *Destroy_pset*, *Invalidate_scheduling_policy*, *Observe_pset_processes*. We use *Procset_control_port_permissions* to denote this set of permissions.

We use *Procset_permissions* to denote the union of *Procset_name_port_permissions* and *Procset_control_port_permissions*.

5.4.13 Device Permissions

The DTOS kernel enforces the following permissions on device requests: *Close_device*, *Control_pager*, *Get_device_status*, *Map_device*, *Open_device*, *Read_device*, *Set_device_filter*, *Set_device_status*, *Write_device*. We use *Device_permissions* to denote this set of permissions.

5.4.14 Kernel Reply Port Permissions

The DTOS kernel enforces the following permissions on requests sent to kernel reply ports: *Provide_permission*. We use *Kernel_reply_permissions* to denote this set of permissions.

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

5.5 Access Vector Cache

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

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

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

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

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

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

5.6 Message Security Information

Each existing message has an SSI associated with it that indicates the SSI of the task that sent the message. The expression $\underline{msg}\_sending\_sid(message)$ indicates the SSI of the task that

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

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

5.7 Task Creation Information

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

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

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

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

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

---

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

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

The Mach model of process creation uses an existing task to serve as a "template" for each new task. This task is the `parent_task` parameter to **task_create**. A newly created task inherits parts of its environment, such as portions of its address space, from the "parent" task. To simplify the statement of the security requirements on task creation, we introduce $\underline{parent\_task}(task)$ to denote $task$'s parent.[9]

5.8 Server Ports

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

- $\underline{security\_server\_master\_port}$ denotes the port used by the kernel to make requests of the security server.
- $\underline{security\_server\_client\_port}$ denotes the port used by non-kernel clients to make requests of the security server.
- $\underline{authentication\_server\_port}$ denotes the port used to make requests of the authentication server.
- $\underline{audit\_server\_port}$ denotes the port used to make requests of the audit server.
- $\underline{crypto\_server\_port}$ denotes the port used to make requests of the crypto server.
- $\underline{negotiation\_server\_port}$ denotes the port used to make requests of the negotiation server.
- $\underline{network\_ss\_port}$ denotes the port used to make security requests over the network.

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

5.9 Memory Region Protections

The current protection of a region limits a task's access to that region. It is calculated as the intersection of the Mach protection together with the accesses allowed for a task to a memory region by the relevant access vector. We use $protection(task, index)$ to denote current protections of the region denoted by a given task-index pair.[10]

5.10 Summary of DTOS Kernel State

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

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

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

Section 6 DTOS Services

This section describes the services provided by DTOS. The organization of this section is as follows:

- Section 6.1 presents a simple execution model for DTOS.
- Sections 6.2– 6.13 define the abstract services relevant to IPC, ports, VM, pagers, threads, tasks, hosts, processors, processor sets, the permissions cache, and devices. Each abstract service is described informally in the context of the model provided in the preceding sections. Each abstract service is assigned a name to facilitate references to the service. The tables in Section 7 use these service names to define the association between services and permissions.
- Section 6.14 defines the abstract services of initiating a kernel outcall. These abstract services are used to state the requirements on kernel outcall services in Section 7.
- Section 6.15 defines the abstract service of initiating an implementation service. This abstract service is used to state the requirements on implementation services in Section 7.

In addition to describing the DTOS services, we also describe security threats that suggest the desirability of controlling the services. Our goal in describing these threats is to provide motivation for our selection of services rather than provide a complete description of all security threats to DTOS.

6.1 Kernel Requests and State Transitions

Editorial Note:
This section provides a very brief execution model for DTOS. Much more detail can be found by consulting the FTLS.

In the simplest model, DTOS kernel state transitions occur as the result of client requests. Requests include the following information:

- *request_op* — The identifier of the operation (such as **mach_port_allocate**) in the request.
- *eff_client* — The client task making the request.
- *service_port* — The port through which the request is received.
- *initial_ruling* — As part of the initial processing of a request, the kernel associates a ruling with the request.

Editorial Note:
When *initial_ruling* is computed, the SSI associated with the ruling is the SID of *eff_client* and the OSI associated with the ruling is the SID of the port through which the request was received (modulo the *Task_self_sid* and *Thread_self_sid* computations). In later processing of the request, the kernel sometimes assumes that

- the SID of *eff_client* is the same as when the request was received,

- the SID of the port through which the request was received is the same as when the request was received, and
- the permissions in $initial\_ruling$ are still valid.

This lack of support for non-tranquility is not reflected in the model.

Editorial Note:
In the current execution model, $Request$ refers to service requests sent through **mach_msg** as well as traps; the fields $request\_op$ and $service\_port$ only apply in the case of a **mach_msg** request.

During the initial processing of a newly received request, the DTOS kernel performs some integrity validation and permission checks. We use $validated\_requests$ to model the collection of requests that have successfully passed these checks. Note that it is possible that some of the requests in $validated\_requests$ are identical.

Each DTOS kernel state transition is associated with the following:

- $client$ — The task responsible for the transition occurring. If the transition is a direct response to a kernel request, then $client$ is the task which made the request (eff_client). Otherwise, $client$ is the kernel task.
- $client\_sid$ — The current SID of $client$. If $client$ no longer exists when its request is being processed, then this is the SID of $client$ when it was destroyed.

  Editorial Note:
  This used to say that it was the SID of $client$ when the request was made, but I do not believe that is true. The spec-to-code analysis should make the determination. We also may need to make the determination of what it *should* be.

  Editorial Note:
  This also needs to get fixed to reflect the possibility of a specified SID when the prototype is updated.

- $rulings$ — Zero or more rulings may be retrieved from the security server during the transition. The $initial\_ruling$ contained in the request might be included in this set, or it might come from the cache. These rulings need not be part of the initial or final state of the transition.
- $kernel\_allows(ssi, osi)$ — This function returns the set of permissions which are allowed either by the kernel's cache or by a ruling associated with the transition.

In addition, transitions are often associated with a particular kernel request.

Editorial Note:
Using $\underline{h}ost\_time$ for $Ruling\_allows$ above is probably not correct. Each ruling might be checked at a different time.

We use the set $Valid\_transitions$ to denote the set of transitions that are possible on the DTOS system.[11]

[11] One of the purposes of the FTLS is to define this set of $Valid\_transitions$.

6.2 IPC Services

Mach IPC consists of sending messages to and receiving messages from ports. Although the Mach capability mechanisms (port rights) provide control over which tasks can send messages to each port, the mechanisms are relatively weak. For example, any task that holds a right may pass the right to any other task. Thus, a "trusted" task that holds a right for a "privileged" port might accidentally transfer the right to a malicious task. The DTOS policy addresses this threat by controlling the transferring of rights and the ability to hold rights (permissions *Transfer_rights*, *Hold_receive*, *Hold_send*, *Hold_send_once*, *Transfer_receive*, *Transfer_send*, and *Transfer_send_once*).

Another weakness of capabilities is that the holding of a right implies the ability to use the right. Tasks such as name servers will need to hold rights to many entities that they do not need to access themselves. This is a violation of "least privilege." The DTOS policy addresses this threat by making a distinction between the ability to hold a right versus use a right (permissions *Hold_receive*, *Hold_send*, and *Hold_send_once* versus *Can_receive* and *Can_send*).

In addition to being able to pass port rights in messages, tasks may also pass regions of memory. Since these memory regions can be backed by untrusted pagers and may consume a lot of space in the receiver's address space, there are integrity and denial of service threats related to such data transfers. The DTOS policy addresses these threats by controlling which tasks can transfer memory regions through messages sent to ports. For example, the policy could be used to prohibit the transfer of memory regions through a service port for a trusted server that provides an interface using only in-line data transfer (permission *Transfer_ool*).

As noted in Section 5, DTOS tags messages with a sending SSI and (potentially) a receiving SSI. Although the kernel protects the tags while the message is in transit, certain tasks must be trusted to override the normal use of these tags. For example, a user space network server interposing on user ports needs to receive messages for which it is not the ultimate receiver and copy the original sender's SSI onto the forwarded message. However, the overriding must be controlled for the tags to serve their purpose. The DTOS policy addresses threats to the correctness of the tags by controlling which tasks are permitted to override the normal processing (permissions *Specify* and *Interpose*).

Service Definition 1 (*InitiatesMsgSend***)** *A state transition initiates the sending of a message to port if the client is not the kernel, there exists a new msg sent to port, and port is not destroyed.*

When the kernel sends a message, it is considered an outcall. Outcall services are defined in Section 6.14.

Editorial Note:
Currently, outcall messages are only distinguished from other messages when computing the Send permission. We need to decide if this is appropriate.

Service Definition 2 (*InitiatesRightsTransfer***)** *A state transition initiates the sending of a message to port and the transfer of port rights in the body of the message if there exists a msg such that:*

- *msg is a new message,*
- *msg's destination is port, and*
- *some element of the body of msg carries a port right.*

Note that this service involves the transferring of port rights in the bodies of messages sent *to port* while the following three services involve the transferring of port rights *for port*.

Service Definition 3 ($InitiatesReceiveTransfer$**)** *A state transition initiates the transfer of a receive right for* $port$ *if there exists a* msg *such that:*

- msg *is a new message, and*
- *some element of the body of* msg *carries a receive right for* $port$.

Service Definition 4 ($InitiatesSendTransfer$**)** *A state transition initiates the transfer of a send right for* $port$ *if there exists a* msg *such that:*

- msg *is a new message, and*
- *the reply port field in the header of* msg *or some element of the body of* msg *carries a send right for* $port$.

Service Definition 5 ($InitiatesSendOnceTransfer$**)** *A state transition initiates the transfer of a send-once right for* $port$ *if there exists a* msg *such that:*

- msg *is a new message, and*
- *the reply port field in the header of* msg *or some element of the body of* msg *carries a send-once right for* $port$.

Service Definition 6 ($InitiatesOolDataTransfer$**)** *A state transition initiates the transfer of out-of-line data through* $port$ *if there exists a* msg *such that:*

- msg *is a new message,*
- msg*'s destination is* $port$*, and*
- *some element of the body of* msg *carries an out-of-line region.*

Service Definition 7 ($SetsReply$**)** *A state transition sets the reply port to which a reply message will be sent to* $port$ *if there exists a* msg *such that:*

- msg *is a new message,*
- msg*'s* $local\_port$ *is* $port$.

Service Definition 8 ($SpecifiesSsi$**)** *A state transition initiates the sending of a message to* $port$ *with a sending SID specified if there exists a* msg *such that:*

- msg *is a new message,*
- msg*'s destination is* $port$*, and*
- msg *is in the domain of the function* $\underline{msg}\_specified\_sid'$.

Service Definition 9 ($SpecifiesAV$**)** *A state transition initiates the sending of a message to* $port$ *with an access vector specified if there exists a* msg *such that:*

- msg *is a new message,*
- msg*'s destination is* $port$*, and*
- msg *is in the domain of the function* $\underline{msg}\_specified\_vector'$.

The remaining IPC services consider the receiving of messages.

Service Definition 10 ($InitiatesMsgReceive$**)** *A state transition initiates the receiving or removal of a message from* $port$ *if there exists* msg *such that:*

- *msg is removed from the queue associated with port, and*
- *port is not destroyed*

Service Definition 11 (*Interposes***)** *A state transition receives a message with a specified receiving SID other than the client's SID from port if there exists a msg such that:*

- *in the initial state, msg is enqueued at port,*
- *msg is added to the set of messages received by client,*
- *a receiving SID is specified for msg, and*
- $\underline{msg}\_receiving\_sid(msg) \neq client\_sid$.

Note that the services *InitiatesMsgReceive* and *Interposes* present different definitions of "receiving" a message. This is because the *InitiatesMsgReceive* considers the more general case of receiving or removing a message from the queue, while *Interposes* only considers the case of receiving a message from the queue. This distinction is important since it may be necessary for a client to remove a message from a queue when it is not allowed to receive the message because it is not the specified receiver.

6.3 Port Services

All kernel entities are represented by ports. Consequently, the security of a task rests on the ability to protect the task's port name space. Mach allows any task holding a send right to a second task's self port to modify the second task's port name space.

By allocating rights in a second task's port name space, a malicious task can consume resources in that task. This could lead to a denial of service. The DTOS policy addresses this by controlling the adding of names to port name spaces (permission *Add_name*).

If a malicious task can deallocate rights in a second task's port name space, then it can make resources the second task is using unavailable. This can lead to a denial of service, too. The DTOS policy addresses this by controlling the removal of names from port name spaces (permission *Remove_name*).

A related threat is the moving of a port right in a port name space. For example, if a malicious task renames a port right or changes the members of a port set in a second task, the second task might fail as the result of port rights no longer being where they should be. The DTOS policy addresses this threat by controlling the moving of names within a port name space (permissions *Port_rename*, *Manipulate_port_set*, *Register_ports*).

A more subtle type of threat is the modification of the notifications registered for a task. If a task has a thread waiting to receive a notification and the notification is canceled by a second task, then the thread will never receive the notification. The DTOS policy addresses this by controlling the registering of notifications (permission *Register_notification*).

Other threats include modifying the make-send count, queue limit, or sequence number for a port. For example, if the queue limit for a server's service port is decreased, messages sent to the server might start to time out. This could result in a denial of service. The DTOS policy addresses such threats by controlling the setting of these port attributes (permission *Alter_pns_info*).

The DTOS policy with respect to MIDs is tranquil in that once the kernel associates a MID with an entity, the entity remains bound to the same MID. For ports, we represent this by defining a service, *ChangesPortMid*, that characterizes the changing of a port's MID and then prohibiting this service in Section 7.

However, the AID of some entities is allowed to change. The AID of a port is allowed to change only in the case of a task or thread port for a task whose AID also changes. We represent this by defining a service, $ChangesPortAid$, that characterizes the changing of a port's AID under all other circumstances, and prohibiting this service in Section 7.

Service Definition 12 ($AddsReceive$**)** *A state transition adds a receive right for* $port$ *to* $task$*'s port name space if* $task$ *obtains a receive right for* $port$ *and* $task$ *did not previously hold a receive right for* $port$*.*[12]

Service Definition 13 ($AddsSendRight$**)** *A state transition adds a send to* $port$ *right to* $task$*'s port name space if* $task$ *obtains a send to* $port$ *right and* $task$ *did not previously hold a send to* $port$*right.*

Service Definition 14 ($AddsSendReference$**)** *A state transition adds a send to* $port$ *reference to* $task$*'s port name space if* $task$ *has a send to* $port$ *right and* $task$ *obtains an additional reference to a send to* $port$ *right.*

Composite Service Definition 1 *We refer to the service in which either a send right is created or a reference count for a send right is incremented as the service* $AddsSend$*.*

Service Definition 15 ($AddsSendOnce$**)** *A state transition adds a sendonce to* $port$ *right to* $task$*'s port name space if* $task$ *obtains a sendonce to* $port$ *right and* $task$ *did not previously hold a sendonce to* $port$ *right.*

Service Definition 16 ($AddsDeadNameRight$**)** *A state transition adds a dead name right to* $task$*'s port name space if the number of names which are dead and not previously in* $task$*'s port name space is greater than the number of names which were dead and are not currently in* $task$*'s port name space.*

Editorial Note:
This service ignores the case where a dead name is created when the corresponding port dies (such a name is not in either of the set comprehensions) as well as the case where a dead name is renamed (the first set contains the new dead name but not the old, while the second set contains the old dead name but not the new.)

Service Definition 17 ($AddsDeadNameReference$**)** *A state transition adds a dead name reference to* $task$*'s port name space if* $task$ *obtains an additional reference to a dead name right that it previously held.*

Composite Service Definition 2 *We refer to the service in which either a dead name is created or a reference count for a dead name is incremented as the service* $AddsDeadName$*.*

Service Definition 18 ($CreatesPortSet$**)** *A state transition creates a new port set in* $task$*'s port name space if the number of entries for* $task$ *in* $port\_set\_namep$*, the set of* $(task, name)$ *pairs that denote port sets, is increased.*

Composite Service Definition 3 *We refer to the service in which either a receive, send, send-once right, a dead name, or a port set is added to* $task$*'s port name space as the service* $AddsName$*.*

Editorial Note:
The next four service definitions are intended to refer to services which explicitly remove rights from a name space. As stated, they are much too broad. For instance,

[12] Note that this service does not define the SID that is associated with the $port$. The expression $port\_sid'(port)$ denotes this SID. The requirements in Section 7 require that the client have permission to add ports with this SID to $task$'s port name space and that $task$ have permission to hold ports with this SID. No other restrictions are placed on the SID assigned to the new port.

- They don't consider the possibility that a send or send-once right disappears because it is used to send a message.
- They don't consider the possibility that a right disappears because it is sent in a message.

As a general rule, side effects of destroying a port aren't handled well in these abstract service definitions. There is a long chain of possible consequences to simply removing one port.

Editorial Note:
It is not clear that we can identify rights that are being removed as a result of being transferred in a message or expiring due to use. At first glance it seems that we can check for the existence of an appropriate message as in the definition of the IPC services $InitiatesRightsTransfer$, $InitiatesReceiveTransfer$, etc. According to the specification of $Transit\_right$ (see the definition of $BASE\_INTERNAL\_ELEMENT$) we can recover the name of the task initiating the transfer of rights from the message contents, but we do not believe that this is carried out in the prototype. An alternative is to compare $Request$'s eff_client with $task$, but the complexity of the execution model makes it difficult to determine if this handles all cases accurately.

Service Definition 19 ($RemovesReceive$**)** *A state transition removes a receive right for* $port$ *from* $task$*'s port name space if* $task$ *loses a receive right for* $port$ *that it previously held, and* $task$ *is not destroyed.*

Service Definition 20 ($RemovesSendRight$**)** *A state transition removes a send to* $port$ *right from* $task$*'s port name space if* $task$ *loses a send to* $port$ *right that it previously held, and* $task$ *and* $port$ *are not destroyed.*

Service Definition 21 ($RemovesSendReference$**)** *A state transition removes a send to* $port$ *reference from* $task$*'s port name space if* $task$*'s reference count for a send right for* $port$ *is decreased, and* $task$ *and* $port$ *are not destroyed.*

Composite Service Definition 4 *We refer to the service in which either a send right is removed or a reference count for a send right is decremented as the service* $RemovesSend$*.*

Service Definition 22 ($RemovesSendOnce$**)** *A state transition removes a sendonce to* $port$ *right from* $task$*'s port name space if* $task$ *loses a sendonce to* $port$ *right that it previously held, and* $task$ *and* $port$ *are not destroyed.*

Service Definition 23 ($RemovesDeadNameRight$**)** *A state transition removes a dead name right from* $task$*'s port name space if the number of dead names in* $task$*'s port name space is decreased, and* $task$ *is not destroyed.*

Service Definition 24 ($RemovesDeadNameReference$**)** *A state transition removes a dead name reference from* $task$*'s port name space if* $task$ *loses a reference to a dead name right that it previously held, and* $task$ *is not destroyed.*

Composite Service Definition 5 *We refer to the service in which either a dead name is destroyed or a reference count for a dead name is decremented as the service* $RemovesDeadName$*.*

Service Definition 25 ($DestroysPortSet$**)** *A state transition destroys a port set in* $task$*'s port name space if the number of names in* $task$*'s port name space that are port set names decreases, and* $task$ *is not destroyed.*

Composite Service Definition 6 *We refer to the service in which either a receive, send, send-once right, a dead name, or a port set is removed from* $task$*'s port name space as the service* $RemovesName$*.*

Service Definition 26 ($RenamesInPortNameSpace$**)** *A state transition renames a port, dead name, or port set in* $task$*'s port name space if an entry is removed from the name space and an identical entry is added under a different name.*

Service Definition 27 ($ManipulatesPortSet$**)** *A state transition manipulates a port set in* $task$*'s port name space if there is some port set* $name$ *such that the set of ports associated with* $name$ *is altered in a manner other than by simply removing ports for which* $task$ *is no longer the receiver. In other words, there exists* $name$ *and* $port$ *such that:*

- $name$ *represents a port set in* $task$*'s port name space in both the initial and final states of the transition, and*
  - $port$ *is added to* $port\_set(task, name)$*, or*
  - $port$ *is removed from* $port\_set(task, name)$ *and* $task$ *remains the receiver from* $port$*.*

Service Definition 28 ($RegistersPort$**)** *A state transition registers a port or removes a previously registered port associated with a task if there exists* $port$ *which*

- *is added to* $\underline{r}egistered\_rights(task)$*, or*
- *is removed from* $\underline{r}egistered\_rights(task)$ *without being destroyed.*

Service Definition 29 ($RegistersPortDestroyedNotification$**)** *A state transition registers a port-destroyed notification request for a port in* $task$*'s name space or removes a previously registered request if there exists* $port_1$ *and* $port_2$ *such that*

- $task$ *is the receiver for* $port_1$ *in both the initial and final states of the transition, and*
  - $port_2$ *is added to* $port\_notify\_destroyed(port_1)$*, or*
  - $port_2$ *is removed from* $port\_notify\_destroyed(port_1)$ *without being destroyed.*

Service Definition 30 ($RegistersNoMoreSendersNotification$**)** *A state transition registers a no-more-senders notification request for a port in* $task$*'s name space or removes a previously registered request if there exists* $port_1$ *and* $port_2$ *such that*

- $task$ *is the receiver for* $port_1$ *in both the initial and final states of the transition, and*
  - $port_2$ *is added to* $port\_notify\_no\_more\_senders(port_1)$*, or*
  - $port_2$ *is removed from* $port\_notify\_no\_more\_senders(port_1)$ *without being destroyed.*

Service Definition 31 ($RegistersDeadNameNotification$**)** *A state transition registers a dead-name notification request for a port in* $task$*'s name space or removes a previously registered request if there exists* $name$ *and* $port$ *such that*

- $name$ *represents some port right in* $task$*'s name space in both the initial and final states of the transition, and*
  - $port$ *is added to* $port\_notify\_dead(task, name)$*, or*
  - $port$ *is removed from* $port\_notify\_dead(task, name)$ *without being destroyed.*

Composite Service Definition 7 *We refer to the service in which either a port-destroyed, no-more-senders, or dead-name notification is registered or removed from* $task$*'s port name space as the service* $RegistersNotification$*.*

Service Definition 32 ($SetsMakeSendCount$**)** *A state transition modifies the make-send count for a port in* $task$*'s port name space if there is some* $port$ *for which* $task$ *is the receiver and* $\underline{m}ake\_send\_count(port)$ *is altered.*

Service Definition 33 ($SetsQueueLimit$**)** *A state transition modifies the queue limit for a port in* $task$*'s port name space if there is some* $port$ *for which* $task$ *is the receiver and* $\underline{q}\_limit(port)$ *is altered.*

Service Definition 34 (*SetsSeqNo*) *A state transition modifies the sequence number for a port in task's port name space if there is some port for which task is the receiver and sequence_no(port) is altered.*

Editorial Note:
Each of the previous three service definitions need to be checked (and changed) for possible side effects. For instance, the make-send count changes as a side effect of creating new rights. I don't believe there are side effect with the queue limit. The sequence number can change whenever a message is removed from a message queue.

Composite Service Definition 8 *We refer to the service in which either the make-send count, queue limit, or sequence number for a port is altered as the service ModifiesPortInfo.*

Service Definition 35 (*ChangesPortMid*) *A state transition changes a port's MID if it alters port_mid(port).*

Service Definition 36 (*ChangesPortAid*) *A state transition changes a port's AID if it alters port_aid(port), except in the case of the AID of a task or thread port changing due to a similar change in the task AID.*

6.4 VM Services

An interesting feature of the Mach Virtual Memory (VM) system is that it allows a task holding a send right to a second task's self port to access the address space of the second task.

By allocating memory in a second task's address space, a malicious task can consume resources in that task. This could lead to a denial of service. The DTOS policy addresses this by controlling the allocation of memory (permission *Allocate_vm_region*).

If a malicious task can deallocate memory in a second task's address space, then it can make memory the second task is using unavailable. This can lead to a denial of service, too. The DTOS policy addresses this by controlling the deallocation of memory (permission *Deallocate_vm_region*).

To protect the integrity and confidentiality of data, it is necessary to control the Mach, current and maximum protections for memory allocated to a task. The DTOS policy addresses these concerns by requiring the permissions cache be consulted whenever protections are set (permissions *Have_read*, *Have_write*, *Have_execute*, *Chg_vm_region_prot*).

A more subtle way in which data integrity or confidentiality can be compromised is through the modification of inheritance attributes. For example, a malicious task might change the inheritance attribute of a memory object that a second task wants to keep private so that the object is shared with children of the second task. The DTOS policy addresses this by controlling the modification of inheritance attributes (permission *Set_vm_region_inherit*).

Service Definition 37 (*AllocatesRegion*) *A state transition allocates a region at page_index in task's virtual address space if page_index is allocated for task in the final state but not in the initial state.*

Service Definition 38 (*AllocatesReadRegion*) *A state transition allocates a readable region at page_index in task's virtual address space if page_index is allocated for task with read access in the final state but not in the initial state.*

Service Definition 39 (*AllocatesWriteRegion*) *A state transition allocates a writable region at page_index in task's virtual address space if page_index is allocated for task with write access in*

the final state but not in the initial state.

Service Definition 40 (*AllocatesExecuteRegion***)** *A state transition allocates an executable region at page_index in task's virtual address space if page_index is allocated for task with execute access in the final state but not in the initial state.*

Editorial Note:
The preceding four services are all performed by the **vm_allocate**, **vm_allocate_secure** and **vm_map** requests. It was realized recently that the FSPM and the prototype contained different checks related to these requests/services and that neither was "correct". This draft of the FSPM contains corrected requirements for these services. The prototype will be modified at a later date to implement these new requirements.

These services are also performed by **mach_msg**. We need to consider what permission checks are required in this case. (It appears that the prototype is checking read, write and execute permissions, but not *Allocate_vm_region* and *Map_vm_region*.)

Service Definition 41 (*DeallocatesRegion***)** *A state transition deallocates a region in task's virtual address space if it decreases the number of pages in task's virtual address space.*

Service Definition 42 (*SetsProtection***)** *A state transition changes the protection of a region in task's virtual address space if there is some page_index for which either mach_protection(task, page_index) or max_protection(task, page_index) is altered.*

Service Definition 43 (*SetsInheritance***)** *A state transition changes the inheritance attributes of a region in task's virtual address space if there is some page_index for which inheritance(task, page_index) is altered.*

Service Definition 44 (*ModifiesRegion***)** *A state transition modifies a memory region in task's virtual address space if there exists memory, page, and page_offset such that page is associated with memory and page_word_rel(page, page_offset) is altered.*

6.5 Pager Services

Mach's support for user pagers introduces threats that are not usually a concern. If a malicious task can act as the pager for an object, it can provide incorrect data for the object or make the object unavailable. The DTOS policy controls which tasks can page a memory (permission *Provide_data*) and which tasks can make an object unavailable (permissions *Destroy_object*, *Change_page_locks*, *Remove_page*).

Examples of more subtle threats are changing the set of precious pages and flushing dirty pages. In both cases, the kernel could fail to make the pager aware of modifications that have been made to the pages. The DTOS policy addresses these threats by controlling which tasks may change the set of precious pages or flush dirty pages (permissions *Make_page_precious*, *Save_page*).

Service Definition 45 (*ChangesMemoryObjectAttr***)** *A state transition changes the attributes of memory if it alters copy_strategy(memory) or adds or removes memory from may_cache.*

Service Definition 46 (*ServicesPageFault***)** *A state transition services a page fault for memory if it removes a thread from the set of threads that memory_fault indicates are waiting on a page from memory.*

Service Definition 47 (*MakesPagePrecious***)** *A state transition makes a page representing memory precious if there is a page representing memory that is added to or removed from precious.*

Service Definition 48 (*ChangesPageLocks***)** *A state transition modifies the locks on a page representing memory if there exists a page representing memory such that page_lock_rel(page) is altered.*

Editorial Note:
The preceding two services (as well as *SavesPage* and *RemovesPage* below) do not currently have any associated policy requirements. We are considering whether the 12 permissions currently defined for memory control services can actually be reduced to a single permission indicating that the subject can serve as the pager for a given memory object. The case for doing this is that any usable pager probably needs to be allowed to use the entire paging protocol. Thus, the ability to page for a memory object may well be an all-or-nothing proposition. If so, nothing is gained by having 12 permissions.

Service Definition 49 (*DestroysMemory***)** *A state transition destroys memory if it removes memory from control_port.*

Service Definition 50 (*SavesPage***)** *A state transition saves a dirty page representing memory if there exists a page representing memory such that page is removed from dirty_rel.*

Service Definition 51 (*RemovesPage***)** *A state transition removes a page representing memory if there exists a page representing memory such that page is removed from represented_memory.*

6.6 Thread Services

The attributes associated with a thread determine if and when a thread may execute. Modification of these attributes can lead to denial of service conditions. For example, if a malicious task depresses the priority of a thread, that thread might be prevented from executing. As another example, a malicious task could prevent a thread from executing by incrementing the thread's suspend count. The DTOS policy addresses such threats by controlling modifications to thread attributes (*Assign_thread_to_pset*, *Set_max_thread_priority*, *Set_thread_policy*, *Terminate_thread*, *Set_thread_priority*, *Depress_pri*).

The resumption of a thread is also a concern. For example, if a thread is resumed before an event that it is waiting on has completed, then the thread might fail to operate correctly. The DTOS policy addresses this threat by controlling which tasks can decrement a thread's suspend count (permissions *Initiate_secure*, *Resume_thread*).

Another threat to threads is that a malicious task can change the thread's sself or exception port. If the sself port is changed before the thread gets a send right to it, then when the thread requests a send right to its kernel port, it is given a right to the sself port instead. When the thread later attempts to send kernel requests to its kernel port, the requests will actually be sent to other ports. If the exception port is changed, then the thread will not receive exception messages and might fail to operate properly. The DTOS policy addresses these threats by controlling the changing of a thread special port (permissions *Set_thread_kernel_port*, *Set_thread_exception_port*).

A more subtle threat is the changing of a thread's program counter. Doing so will change the location in memory from which the thread is reading instructions. Problems that could occur include the thread attempting an illegal instruction and failing or the thread skipping over a section of its code that performs some security check. The DTOS policy addresses this threat by controlling which tasks have access to a thread's register set (permission *Set_thread_state*).

Service Definition 52 (*DepressesPriority***)** *A state transition depresses thread's priority if it adds thread to depressed_threads, the set of depressed threads.*

Service Definition 53 (*AbortsPriorityDepression***)** *A state transition aborts the depression of thread's priority if it removes thread from* $\underline{d}epressed\_threads$*, the set of depressed threads.*

Service Definition 54 (*AssignsThread***)** *A state transition assigns thread to procset if it adds thread to* $have\_assigned\_threads(procset)$*, the set of threads assigned to procset.*

Service Definition 55 (*ResumesThread***)** *A state transition resumes thread in the normal Mach paradigm if it decrements* $\underline{t}hread\_suspend\_count(thread)$ *without changing the task create state of the associated task.*

Service Definition 56 (*MakesTaskReady***)** *A state transition resumes thread in the DTOS cross-context-create paradigm if it decrements* $\underline{t}hread\_suspend\_count(thread)$ *and changes the task create state of the associated task to* Tcs_task_ready*. Note that this service is a DTOS enhancement.*

Service Definition 57 (*IncrementsThreadMaxPriority***)** *A state transition increments thread's maximum priority if* $\underline{t}hread\_max\_priority(thread)$ *is incremented and thread assignments do not change.*

Service Definition 58 (*DecrementsThreadMaxPriority***)** *A state transition decrements thread's maximum priority if* $\underline{t}hread\_max\_priority(thread)$ *is decremented and thread assignments do not change.*

Editorial Note:
Care must be taken in mapping the prior two services to the implementation. The higher the numeric value of a priority, the lower the priority. Thus, incrementing a priority is a decrease in priority while decrementing a priority is an increase in priority.

Service Definition 59 (*SetsThreadPriority***)** *A state transition sets thread's priority if* $\underline{t}hread\_priority(thread)$ *is altered,* $\underline{t}hread\_max\_priority(thread)$ *does not change, the depression status of no thread changes and thread assignments do not change.*

Service Definition 60 (*SetsThreadPolicy***)** *A state transition sets thread's policy if* $\underline{t}hread\_sched\_policy(thread)$ *is altered and thread assignments have not changed.*

Service Definition 61 (*SetsThreadKernelPort***)** *A state transition sets thread's kernel port if it alters* $thread\_sself(thread)$*, thread's kernel port. Note that* $thread\_sself(thread)$ *may be undefined in either the current or new state.*[13]

Service Definition 62 (*SetsThreadExceptionPort***)** *A state transition sets thread's exception port if it alters* $thread\_eport(thread)$*, thread's exception port. Note that* $thread\_eport(thread)$ *may be undefined in either the current or new state.*

Service Definition 63 (*MakesThreadOwnerReady***)** *A state transition sets thread's machine state in the DTOS cross-context-create paradigm if* $\underline{t}hread\_state(thread)$ *is altered and the task creation state of the associated task is set to* $Tcs\_thread\_state\_set$*. Note that this service is a DTOS enhancement.*

Service Definition 64 (*SuspendsThread***)** *A state transition suspends thread if it increments* $\underline{t}hread\_suspend\_count(thread)$*.*

Service Definition 65 (*TerminatesThread***)** *A state transition terminates thread if it removes thread from* $\underline{t}hread\_exists$*, the set of existing threads without removing its parent task from* $\underline{t}ask\_exists$*.*

[13] The expression $R(\!|S|\!)$ denotes the relational image of the set S under relation R (i.e., the set of all values to which an element of S is mapped by R).

Service Definition 66 (*EnablesThreadSampling***)** *A state transition enables sampling for thread if thread is added to sampled_threads, the set of threads currently being sampled.*

Service Definition 67 (*DisablesThreadSampling***)** *A state transition disables sampling for thread if thread is removed from sampled_threads, the set of threads currently being sampled.*

6.7 Task Services

Many of the task services are analogous to thread services. For example, there is a task suspend service that is analogous to the thread suspend service. Due to the similarity of the requests, there are similar threats to be addressed and the DTOS policy addresses those threats using task permissions analogous to the thread permissions used to address the thread services.

An example of a threat that is specific to tasks is the manipulation of emulation vectors. If correct operation of a task requires it use some emulation library, then the task can be caused to fail by modifying its emulation vector. The DTOS policy addresses this threat by controlling which tasks are allowed to modify each task's emulation vector (permission *Set_emulation*).

The DTOS policy with respect to MIDs is tranquil in that once the kernel associates a MID with an entity, the entity remains bound to the same MID. For tasks, we represent this by defining a service, *ChangesTaskMid*, that characterizes the changing of a task's MID and then prohibiting this service in Section 7.

However, the AID of a task may be allowed to change. A service, *ChangesTaskAid*, is defined to characterize the changing of a task's AID.

Service Definition 68 (*AddsThread***)** *A state transition adds a thread to task in the normal Mach paradigm if it adds a thread to threads(task), the set of threads belonging to task, and does not change the task creation state of task.*

Service Definition 69 (*AddsThreadSecure***)** *A state transition adds a thread to task in the DTOS cross-context-create paradigm if it adds a thread to threads(task), the set of threads belonging to task, and changes the task creation state of task to Tcs_thread_created. Note that this service is a DTOS enhancement.*

Service Definition 70 (*AssignsTask***)** *A state transition assigns an existing task to procset if it adds task to the set have_assigned_tasks(procset).*

Service Definition 71 (*SetsTaskPriority***)** *A state transition sets task's priority if it changes the value of task_priority(task).*

Service Definition 72 (*CreatesTask***)** *A state transition creates child in the normal Mach paradigm if it adds child to task_exists and sets child's task creation state to Tcs_task_ready.*

Service Definition 73 (*CreatesTaskSecure***)** *A state transition creates child in the DTOS cross-context-create paradigm if it adds child to task_exists and sets child's task creation state to Tcs_task_empty. Note that this service is a DTOS enhancement.*

Service Definition 74 (*InvTaskCreationStateTrans***)** *A state transition changes task's creation state inappropriately if it does not follow the pattern non-existent → Tcs_task_ready or the pattern non-existent → Tcs_task_empty → Tcs_thread_created → Tcs_thread_state_set → Tcs_task_ready.*

Service Definition 75 (*ResumesTask***)** *A state transition resumes task if it decrements task_suspend_count(task).*

Service Definition 76 (*SetsEmulationVector***)** *A state transition sets an emulation vector for task if it alters emulation_vector(task).*

Service Definition 77 (*SuspendsTask***)** *A state transition suspends task if it increments* $\underline{t}ask\_suspend\_count(task)$.

Service Definition 78 (*SetsTaskKernelPort***)** *A state transition sets task's kernel port if* $task\_sself(task)$ *is altered. Note that* $task\_sself(task)$ *may be undefined in either the current or new state.*[14]

Service Definition 79 (*SetsTaskExceptionPort***)** *A state transition sets task's exception port if* $task\_eport(task)$ *is altered. Note that* $task\_eport(task)$ *may be undefined in either the current or new state.*

Service Definition 80 (*SetsTaskBootPort***)** *A state transition sets task's boot port if* $task\_bport(task)$ *is altered. Note that* $task\_bport(task)$ *may be undefined in either the current or new state.*

Service Definition 81 (*TerminatesTask***)** *A state transition terminates task if it removes task from* $\underline{t}ask\_exists$.

Service Definition 82 (*EnablesTaskSampling***)** *A state transition enables sampling for task if task is added to* $\underline{s}ampled\_tasks$*, the set of tasks currently being sampled.*

Service Definition 83 (*DisablesTaskSampling***)** *A state transition disables sampling for task if task is removed from* $\underline{s}ampled\_tasks$*, the set of tasks currently being sampled, and task still exists.*

Service Definition 84 (*ChangesTaskMid***)** *A state transition changes a task's MID if it alters* $task\_mid(task)$.

Service Definition 85 (*ChangesTaskAid***)** *A state transition changes a task's AID if it alters* $task\_aid(task)$.

6.8 Host Name Port Services

Mach allows tasks holding a send right to the host name port to create new processor sets. Such tasks might be able to cause a resource exhaustion condition by creating a large number of processor sets. The DTOS policy addresses this threat by controlling which tasks are allowed to create processor sets (permission $Create\_pset$).

The DTOS prototype services requests to remove permission sets from the permissions cache through the host name port. Tasks that can remove entries from the cache can deny service by revoking access to resources. The DTOS policy addresses this threat by controlling which tasks can remove entries from the permissions cache (permission $Flush\_permission$).

The DTOS prototype also services requests to change the Security Server Port through the host name port. Since this is the port that indicates where the kernel should send requests for access computations, the security of the system can be compromised if it is set inappropriately. The DTOS policy addresses this threat by controlling which tasks can alter the Security Server Port (permission $Set\_security\_master\_port$).

Service Definition 86 (*CreatesProcset***)** *A state transition creates a processor set procset if procset is added to* $\underline{p}rocset\_exists$.

Service Definition 87 (*FlushesCache***)** *A state transition flushes an entry from the kernel's permission cache if it removes a ruling from* $\underline{c}ache$ *that has not yet expired.*

Service Definition 88 (*SetsSecServerMasterPort***)** *A state transition changes the Security Server master port if it alters the value of* $\underline{s}ecurity\_server\_master\_port$.

[14] The expression $R(\!|S|\!)$ denotes the relational image of the set S under relation R (i.e., the set of all values to which an element of S is mapped by R).

Service Definition 89 (*SetsSecServerClientPort***)** *A state transition changes the Security Server client port if it alters the value of security_server_client_port.*

Service Definition 90 (*SetsAuthenticationServer***)** *A state transition changes the Authentication Server Port if it alters the value of authentication_server_port.*

Service Definition 91 (*SetsAuditServer***)** *A state transition changes the Audit Server Port if it alters the value of audit_server_port.*

Service Definition 92 (*SetsCryptoServer***)** *A state transition changes the Crypto Server Port if it alters the value of crypto_server_port.*

Service Definition 93 (*SetsNegotiationServer***)** *A state transition changes the Negotiation Server Port if it alters the value of negotiation_server_port.*

Service Definition 94 (*SetsNetworkSecurityServer***)** *A state transition changes the Network Security Server Port if it alters the value of network_ss_port.*

Composite Service Definition 9 *A state transition sets a special port if it performs one of the services SetsAuditServer, SetsAuthenticationServer, SetsSecServerClientPort, SetsSecServerMasterPort, SetsCryptoServer, SetsNegotiationServer* or *SetsNetworkSecurityServer.*

6.9 Host Control Port Services

Mach allows tasks holding a send right to the host control port to change the value of the system clock. For applications that use time stamps to ensure consistency, the changing of time can lead to integrity concerns. The DTOS policy addresses this threat by controlling which tasks can change the value of the system clock (permission *Set_time*).

The host control port can also be used to change the default memory manager port recorded by the kernel. If the receiver for the new port does not provide the same functionality as the "real" default manager, then temporary objects would no longer be paged properly. The DTOS policy addresses this threat by controlling which tasks can change the default manager port (permission *Set_default_memory_mgr*).

Another privileged operation that can be performed through the host control port is the wiring of threads into memory. If arbitrary tasks are permitted to wire threads, then the kernel resources can easily be exhausted. The DTOS policy addresses this threat by controlling which tasks are permitted to wire threads (permission *Wire_thread*).

Note that the Mach approach to controlling privileged host operations is to limit the tasks that hold a send right to the host control port. However, there is always the possibility that a task that holds such a right might accidentally transfer it to an inappropriate task. Furthermore, it is not necessarily true that every task that needs to execute a privileged host operation needs to execute all privileged host operations. The DTOS approach addresses the first problem by separating the holding of a right from the ability to use the right. For example, a task must have *Set_time* permission in addition to holding a send right to the host control port to set the system clock.[15] The DTOS approach addresses the second problem by using a separate permission to control each service. For example, a task might be permitted to wire threads while not being permitted to load entries into the permissions cache.

Service Definition 95 (*ChangesWiring***)** *A state transition wires or unwires memory in task's address space if there exists a page_index such that*

[15] This permission is enforced on an implementation service rather than an abstract service since the time may change during virtually any system transition. The prototype prevents any change (increasing or decreasing) via the request **host_set_time** if *Set_time* permission is not held.

- $\underline{w}ire\_count(task, page\_index)$ *is altered, and*
- *the portions of memory that are allocated do not change.*

Note that this might have no effect on the wiring of the page to which the $page\_index$ *is mapped since the page might be wired for a different memory region or it might be wired multiple times for the region affected by this transition.*

Service Definition 96 (*$SetsDefaultManager$***)** *A state transition sets the system's default manager if it alters* $\underline{d}efault\_mem\_manager$.

Service Definition 97 (*$WiresThread$***)** *A state transition wires or unwires* $thread$ *if it adds or removes* $thread$ *from* $\underline{t}hreads\_wired$.

6.10 Processor Services

Mach allows tasks to change the execution status of a processor. For example, processing on a processor can be stopped by "exiting" the processor. As another example, a processor can be moved into a different processor set that schedules threads differently than the processor's initial processor set. The DTOS policy addresses these threats by controlling which tasks can perform operations on processors (permissions $Assign\_processor\_to\_set$, $May\_control\_processor$).

As with privileged host operations, Mach controls these operations through the use of capabilities. Once again, the DTOS control mechanisms are much stronger and more flexible (see the discussion in Section 6.9).

Service Definition 98 (*$AssignsProcessor$***)** *A state transition assigns a processor* $proc$ *to a processor set* $procset$ *if* $proc$ *is added to* $processors(procset)$.

Service Definition 99 (*$ExitsProcessor$***)** *A state transition causes a processor* $proc$ *to be exited if it removes it from the processor set specified by* $proc\_assigned\_procset(proc)$ *and does not add* $proc$ *to any other processor set.*

6.11 Processor Set Control Port Services

Mach allows a task holding a send right to a processor set control port to destroy the processor set. This can adversely impact the scheduling of threads executing on the processor set. Similar effects can also be achieved by changing the scheduling policies supported by the processor set or the maximum priority allowed for threads assigned to the processor set. The DTOS policy addresses these threats by controlling which tasks may operate on a processor set (permissions $Destroy\_pset$, $Chg\_pset\_max\_pri$, $Invalidate\_scheduling\_policy$, $Define\_new\_scheduling\_policy$).

As with privileged host operations, Mach controls these operations through the use of capabilities. Once again, the DTOS control mechanisms are much stronger and more flexible (see the discussion in Section 6.9).

Service Definition 100 (*$DestroysProcset$***)** *A state transition destroys a processor set* $procset$ *if* $procset$ *is removed from* $\underline{p}rocset\_exists$.

Service Definition 101 (*$SetsProcsetMaxPriority$***)** *A state transition sets the maximum scheduling priority for the processor set* $procset$ *if it alters* $\underline{p}s\_max\_priority(procset)$.

Service Definition 102 (*$DisablesPolicy$***)** *A state transition disables a scheduling policy for the processor set* $procset$ *if it removes an element from* $\underline{e}nabled\_sp(procset)$.

Service Definition 103 (*EnablesPolicy***)** *A state transition enables a scheduling policy for the processor set* $procset$ *if it adds an element to* $\underline{e}nabled\_sp(procset)$.

6.12 Kernel Reply Services

The prototype uses kernel reply ports as the service ports for requests to add permission sets to the permissions cache. Tasks that can add entries to the cache can circumvent the DTOS policy. Tasks that can remove entries from the cache can deny service by revoking access to resources. The DTOS policy addresses this threat by controlling which tasks can add entries to the permissions cache (permissions $Provide\_permission$).

Service Definition 104 (*LoadsCache***)** *A state transition loads an entry into the kernel's permission cache if it adds a ruling to* $\underline{c}ache$.

Editorial Note:
We need to consider how to relate the $ruling$ to $kernel\_reply\_port$ in the above.

6.13 Device Services

Tasks in Mach must first open or map a device before accessing the device. Confidentiality can be compromised if a task can open devices that are being used to input data that is inappropriate for the task. Integrity can be compromised if a task can output data through a device that a user believes is being controlled by a trusted task. Availability can be compromised if a task opens a device that only allows a single connection at a time. DTOS protects against these threats by controlling which tasks can open and map each device (permissions $Open\_device$, Map_device). Since service can also be denied by inappropriately closing a device, DTOS controls the closing of devices (permission $Close\_device$). Further protection is provided by controlling the transfer of data through open devices (permissions $Read\_device$, $Write\_device$).

More subtle attacks could be mounted by inappropriately setting the status of a device or the filter associated with a device. For example:

- Packets received through a device could be routed to an inappropriate port as a result of that port being specified as the destination for a filter.
- Packets might not be delivered as they should be due to a filter being changed.

DTOS protects against these threats by controlling the setting of device status and device filters (permissions Set_device_filter, Set_device_status).

Service Definition 105 (*ClosesDevice***)** *A state transition closes* dev *if it decrements* $\underline{d}evice\_open\_count(dev)$, *the count of the number of times that* dev *has been opened and not closed.*

Service Definition 106 (*DecreasesEventCounter***)** *A state transition decreases an event counter* evc *supplied by* dev *if* evc *is supplied by* dev *before and after the transition and the count associated with* evc *decreases.*

Editorial Note:
The service $DecreasesEventCounter$ is not currently controlled in DTOS, but the addition of controls on this service as indicated in Section 7 are planned.

Service Definition 107 (*MapsDevice***)** *A state transition maps dev if it adds dev to $\underline{m}apped\_devices$, the set of mapped devices.*

Service Definition 108 (*OpensDevice***)** *A state transition opens dev if it increments $\underline{d}evice\_open\_count(dev)$, the count of the number of times that dev has been opened and not closed.*

Service Definition 109 (*ReadsDevice***)** *A state transition reads dev if it removes a record from $\underline{d}evice\_in(dev)$, the set of data records input through dev and not yet received.*

Service Definition 110 (*SetsDeviceFilter***)** *A state transition "sets" a filter associated with dev if it changes $\underline{d}evice\_filter\_info(dev)$, the filter information associated with dev.*

Service Definition 111 (*SetsDeviceStatus***)** *A state transition changes the status associated with dev if it changes $\underline{d}evice\_status(dev)$, the status information associated with dev.*

Service Definition 112 (*WritesDevice***)** *A state transition writes dev if it adds a record to $\underline{d}evice\_out(dev)$, the set of data records output through dev and not yet delivered.*

6.14 Outcall Services

A kernel outcall occurs when the kernel sends a message to a port. We define abstract services for the outcalls that the kernel may make. These outcall services must be controlled since a task may cause an outcall to occur by making requests to the kernel. For example, when a task makes a kernel request it may direct the reply message to a given port. When the kernel processing of the request is finished, the kernel will perform an outcall, sending the reply message to the designated reply port. Thus, the task that made the original request has caused a message to be sent to the port. Even though the kernel is sending the message, we want to make sure the task that made the request has permission to send a message to the reply port.

From the kernel's perspective, all of the outcalls require a check of permission Can_send.[16] For outcalls that send reply messages to kernel requests, send exception messages or send a port notification, a non-kernel task must have the permission to send to the destination port of the outcall.

There are several other types of outcall:

security fault — The kernel sends a message to the security server requesting an access vector computation.
page fault — The kernel sends a message to a pager via a memory object's pager port.
pageout — The kernel's pageout daemon determines that a page must be paged out.
send audit information — The kernel sends audit information to an audit port.
forward network packet — The kernel forwards a network packet to a port for the UNIX server.

In contrast to the outcalls described earlier, we require for these types of outcall that the kernel itself have Can_send permission to the destination port of the outcall message. Any effective client on whose behalf the kernel is executing need not have any permission to the destination port. This allows us to restrict the set of tasks that have Can_send permission for important ports such as the security server port. In the case of a page out outcall, the kernel is in fact acting on its own behalf and therefore needs Can_send permission.

[16] For several of the outcalls the server to which the outcall is sent should check additional permissions. As an example, when the security server receives a request to compute an access vector, it checks that the client has permission to make that request. These checks are server-dependent and we do not specify them at this time.

Service Definition 113 (*MakesSecurityOutcall***)** *A state transition makes a security outcall if the kernel sends a request to the security server port with operation* $SSI\_compute\_av\_id$.

Service Definition 114 (*SendsPagerOutcall***)** *A state transition sends a pager outcall if the kernel sends a request with an operation in the set* $Pager\_request\_ids$ *to the object (pager) port of a memory.*

Service Definition 115 (*ConfirmsKernelMemOp***)** *A state transition confirms a memory operation by the kernel if the kernel sends a message with an operation in the set* $Mem\_obj\_confirmation\_ids$. *This message is sent to a reply port provided by the memory manager in an earlier kernel request to which this outcall is the reply.*

Service Definition 116 (*RaisesExceptionToThread***)** *A state transition raises an exception to a thread if the kernel sends a message to the exception port of the thread with operation* $Mach\_exception\_id$.

Service Definition 117 (*RaisesExceptionToTask***)** *A state transition raises an exception to a task if the kernel sends a message to the exception port of the task with operation* $Mach\_exception\_id$.

Service Definition 118 (*SendsKernelReply***)** *A state transition sends a reply from a kernel service request to a reply port if the kernel sends a message to the reply port with an operation in the set* $Kernel\_service\_reply\_ids$.

Service Definition 119 (*SendsNotification***)** *A state transition sends a notification message to a port if the kernel sends a message to the port with an operation in the set* $Mach\_notify\_ids$.

Service Definition 120 (*SendsAuditData***)** *A state transition sends audit data if the kernel sends a message to the* $audit\_server\_port$ *with an operation in the set* $Audit\_ids$.

Service Definition 121 (*ForwardsNetworkPacket***)** *A state transition forwards a network packet to port if the kernel sends a message to a port with the operation set to* $Forward\_net\_packet\_id$.

6.15 Implementation Services

Service Definition 122 (*InitiatesOperation***)** *A state transition initiates* op *if there is a* $request$ *for* op *which is added to* $validated\_requests$.

The set of requests that are treated as implementation services and the permissions associated with these requests are identified in the implementation service tables in Section 7. The DTOS policy addresses each implementation service by only allowing it to be executed when the client holds the permission that the table identifies for the service. The DTOS request **task_get_special_port** can perform any one of three services — getting the kernel, exception or bootstrap port of the task — depending upon the value of one of its parameters. We identify each of these services separately. Similar statements apply to the **thread_get_special_port** and **host_get_special_port** requests.

Section 7

Base Kernel Policy

This section identifies which permission governs each service and which SSI and OSI are used to perform the permissions check. The security policy is simply that a service is only permitted when the permission associated with the service is recorded as being appropriate for the identified SSI and OSI. This section is organized by SSI-OSI pairs. For each SSI-OSI pair, we provide a table identifying the relevant services and their associated permissions.

Note that primed terms are used to indicate values in the state following a transition. For example, $\underline{port\_sid}'(port)$ denotes the SID that *port* will have after the transition rather than *port*'s SID before the transition. This notation is used to state requirements on the creation of entities. For example, when a port is being created, it has no SID in the initial state. Specifying a check on $\underline{port\_sid}'(port)$ indicates that a check should be performed on the SID that *port* will have after it is created.

7.1 Requirements on $client$ to $\underline{port\_sid}'(device\_port'(dev))$ Accesses

Abstract Service	Required Permission
$OpensDevice(dev)$	$Open\_device$

7.2 Requirements on $client$ to $\underline{port\_sid}'(task\_self'(child))$ Accesses

Abstract Service	Required Permission
$CreatesTaskSecure(child)$	$Cross\_context\_create$

7.3 Requirements on $client$ to $\underline{port\_sid}'(task\_self'(task))$ Accesses

Abstract Service	Required Permission
$ChangesTaskAid(task)$	$Make\_sid$

7.4 Requirements on *client* to $\underline{p}ort\_sid(device\_port(dev))$ Accesses

Abstract Service	Required Permission
ClosesDevice(*dev*)	*Close_device*
DecreasesEventCounter(*dev*)	*Wait_evc*
MapsDevice(*dev*)	*Map_device*
ReadsDevice(*dev*)	*Read_device*
SetsDeviceFilter(*dev*)	*Set_device_filter*
SetsDeviceStatus(*dev*)	*Set_device_status*
WritesDevice(*dev*)	*Write_device*

7.5 Requirements on *client* to $\underline{p}ort\_sid(\underline{h}ost\_control\_port)$ Accesses

Abstract Service	Required Permission
ChangesWiring(*task*)	*Wire_vm*
SetsDefaultManager	*Set_default_memory_mgr*
WiresThread(*thread*)	*Wire_thread*

7.6 Requirements on *client* to $\underline{p}ort\_sid(\underline{h}ost\_name\_port)$ Accesses

Abstract Service	Required Permission
CreatesProcset(*procset*)	*Create_pset*
FlushesCache	*Flush_permission*
SetsAuditServer	*Set_audit_port*
SetsAuthenticationServer	*Set_authentication_port*
SetsCryptoServer	*Set_crypto_port*
SetsNegotiationServer	*Set_negotiation_port*
SetsNetworkSecurityServer	*Set_network_ss_port*
SetsSecServerClientPort	*Set_security_client_port*
SetsSecServerMasterPort	*Set_security_master_port*
SetsSpecialPort	*Set_special_port*

7.7 Requirements on *client* to $\underline{p}ort\_sid(kernel\_reply\_port)$ Accesses

Abstract Service	Required Permission
LoadsCache(*kernel_reply_port*)	*Provide_permission*

7.8 Requirements on *client* to $\underline{p}ort\_sid(control\_port(memory))$ Accesses

Abstract Service	Required Permission
ChangesMemoryObjectAttr(*memory*)	*Set_attributes*
DestroysMemory(*memory*)	*Destroy_object*
ServicesPageFault(*memory*)	*Provide_data*

7.9 Requirements on *client* to $\underline{p}ort\_sid(port)$ Accesses

Abstract Service	Required Permission
InitiatesMsgReceive(*port*)	*Can_receive*
InitiatesMsgSend(*port*)	*Can_send*
InitiatesOolDataTransfer(*port*)	*Transfer_ool*
InitiatesReceiveTransfer(*port*)	*Transfer_receive*
InitiatesRightsTransfer(*port*)	*Transfer_rights*
InitiatesSendOnceTransfer(*port*)	*Transfer_send_once*
InitiatesSendTransfer(*port*)	*Transfer_send*
Interposes(*port*)	*Interpose*
SetsReply(*port*)	*Set_reply*
SpecifiesAV(*port*)	*Specify*
SpecifiesSsi(*port*)	*Specify*

7.10 Requirements on *client* to $\underline{p}ort\_sid(proc\_self(proc))$ Accesses

Abstract Service	Required Permission
AssignsProcessor(*proc*,*procset*)	*Assign_processor_to_set*
ExitsProcessor(*proc*)	*May_control_processor*

7.11 Requirements on *client* to $\underline{p}ort\_sid(procset\_self(procset))$ Accesses

Abstract Service	Required Permission
AssignsProcessor(*proc*,*procset*)	*Assign_processor*
AssignsTask(*task*,*procset*)	*Assign_task*
AssignsThread(*thread*,*procset*)	*Assign_thread*
DestroysProcset(*procset*)	*Destroy_pset*
DisablesPolicy(*procset*)	*Invalidate_scheduling_policy*
EnablesPolicy(*procset*)	*Define_new_scheduling_policy*
SetsProcsetMaxPriority(*procset*)	*Chg_pset_max_pri*

7.12 Requirements on *client* to $task\_target(client, \underline{p}arent\_task'(child))$ Accesses

Abstract Service	Required Permission
CreatesTask(*child*)	*Create_task*
CreatesTaskSecure(*child*)	*Create_task_secure*

7.13 Requirements on *client* to $task\_target(client, task)$ Accesses

Abstract Service	Required Permission
AddsName(*task*,*port*)	*Add_name*
AddsThread(*task*)	*Add_thread*
AddsThreadSecure(*task*)	*Add_thread_secure*
AllocatesRegion(*task*,*page_index*)	*Allocate_vm_region*
AssignsTask(*task*,*procset*)	*Assign_task_to_pset*
ChangesWiring(*task*)	*Wire_vm_for_task*
ChangesTaskAid(*task*)	*Change_sid*
DeallocatesRegion(*task*)	*Deallocate_vm_region*
DisablesTaskSampling(*task*)	*Sample_task*
EnablesTaskSampling(*task*)	*Sample_task*
ManipulatesPortSet(*task*)	*Manipulate_port_set*
ModifiesPortInfo(*task*)	*Alter_pns_info*
ModifiesRegion(*task*)	*Write_vm_region*
RegistersNotification(*task*)	*Register_notification*
RegistersPort(*task*)	*Register_ports*
RemovesName(*task*,*port*)	*Remove_name*
RenamesInPortNameSpace(*task*)	*Port_rename*
ResumesTask(*task*)	*Resume_task*
SetsEmulationVector(*task*)	*Set_emulation*
SetsInheritance(*task*)	*Set_vm_region_inherit*
SetsProtection(*task*)	*Chg_vm_region_prot*
SetsTaskBootPort(*task*)	*Set_task_boot_port*
SetsTaskExceptionPort(*task*)	*Set_task_exception_port*
SetsTaskKernelPort(*task*)	*Set_task_kernel_port*
SetsTaskPriority(*task*)	*Chg_task_priority*
SuspendsTask(*task*)	*Suspend_task*
TerminatesTask(*task*)	*Terminate_task*

7.14 Requirements on *client* to *thread_target(client, thread)* Accesses

Abstract Service	Required Permission
AbortsPriorityDepression(thread)	*Abort_thread_depress*
AssignsThread(thread,procset)	*Assign_thread_to_pset*
DecrementsThreadMaxPriority(thread)	*Set_max_thread_priority*
DepressesPriority(thread)	*Depress_pri*
DisablesThreadSampling(thread)	*Sample_thread*
EnablesThreadSampling(thread)	*Sample_thread*
IncrementsThreadMaxPriority(thread)	*Set_max_thread_priority*
MakesTaskReady(thread)	*Initiate_secure*
MakesThreadOwnerReady(thread)	*Initiate_secure*
ResumesThread(thread)	*Resume_thread*
SetsThreadExceptionPort(thread)	*Set_thread_exception_port*
SetsThreadKernelPort(thread)	*Set_thread_kernel_port*
SetsThreadPolicy(thread)	*Set_thread_policy*
SetsThreadPriority(thread)	*Set_thread_priority*
SuspendsThread(thread)	*Suspend_thread*
TerminatesThread(thread)	*Terminate_thread*
WiresThread(thread)	*Wire_thread_into_memory*

7.15 Requirements on *kernel* to *port_sid(audit_server_port)* Accesses

Abstract Service	Required Permission
SendsAuditData	*Can_send*

7.16 Requirements on *kernel* to *port_sid(object_port(memory))* Accesses

Abstract Service	Required Permission
SendsPagerOutcall(memory)	*Can_send*

7.17 Requirements on *kernel* to *port_sid(port)* Accesses

Abstract Service	Required Permission
ConfirmsKernelMemOp(port)	*Can_send*
ForwardsNetworkPacket(port)	*Can_send*

7.18 Requirements on *kernel* to *port_sid(security_server_master_port)* Accesses

Abstract Service	Required Permission
MakesSecurityOutcall	*Can_send*

7.19 Requirements on $kernel\_as(eff\_client)$ to $\underline{p}ort\_sid(port)$ Accesses

Abstract Service	Required Permission
$SendsKernelReply(port)$	Can_send
$SendsNotification(port)$	Can_send

7.20 Requirements on $kernel\_as(eff\_client)$ to $\underline{p}ort\_sid(task\_eport(task))$ Accesses

Abstract Service	Required Permission
$RaisesExceptionToTask(task)$	Can_send

7.21 Requirements on $kernel\_as(eff\_client)$ to $\underline{p}ort\_sid(thread\_eport(thread))$ Accesses

Abstract Service	Required Permission
$RaisesExceptionToThread(thread)$	Can_send

7.22 Requirements on $\underline{p}arent\_task'(child)$ to $\underline{p}ort\_sid'(task\_self'(child))$ Accesses

Abstract Service	Required Permission
$CreatesTaskSecure(child)$	$Cross\_context\_inherit$

7.23 Requirements on $task$ to $\underline{p}age\_sid(task, page\_index)$ Accesses

Abstract Service	Required Permission
$AllocatesExecuteRegion(task,page\_index)$	$Have\_execute$
$AllocatesReadRegion(task,page\_index)$	$Have\_read$
$AllocatesRegion(task,page\_index)$	Map_vm_region
$AllocatesWriteRegion(task,page\_index)$	$Have\_write$

7.24 Requirements on $task$ to $\underline{p}ort\_sid'(port)$ Accesses

Abstract Service	Required Permission
$AddsReceive(task,port)$	$Hold\_receive$

7.25 Requirements on $task$ to $\underline{p}ort\_sid'(task\_self'(task))$ Accesses

Abstract Service	Required Permission
$ChangesTaskAid(task)$	$Transition\_sid$

7.26 Requirements on $task$ to $\underline{p}ort\_sid(port)$ Accesses

Abstract Service	Required Permission
$AddsReceive(task,port)$	$Hold\_receive$
$AddsSend(task,port)$	$Hold\_send$
$AddsSendOnce(task,port)$	$Hold\_send\_once$

7.27 Prohibited Actions on $port$

No transition may perform any of the following services: $ChangesPortAid$, $ChangesPortMid$.

7.28 Prohibited Actions on $task$

No transition may perform any of the following services: $ChangesTaskMid$, $InvTaskCreationStateTrans$.

7.29 Requirements on $client$ to dev Implementation Accesses

Request	Required Permission
device_get_status	Get_device_status

7.30 Requirements on $client$ to $\underline{h}ost\_control\_port$ Implementation Accesses

Request	Required Permission
host_adjust_time	Set_time
host_get_boot_info	Get_boot_info
host_processor_set_priv	$Pset\_ctrl\_port$
host_processors	$Get\_host\_processors$
host_reboot	$Reboot\_host$
host_set_time	Set_time

7.31 Requirements on *client* to *host_name_port* Implementation Accesses

Request	Required Permission
host_get_audit_port	*Get_audit_port*
host_get_authentication_port	*Get_authentication_port*
host_get_crypto_port	*Get_crypto_port*
host_get_host_control_port	*Get_host_control_port*
host_get_negotiation_port	*Get_negotiation_port*
host_get_network_ss_port	*Get_network_ss_port*
host_get_sec_server_client_port	*Get_security_client_port*
host_get_sec_server_port	*Get_security_master_port*
host_get_special_port	*Get_special_port*
host_get_time	*Get_time*
host_info	*Get_host_info*
host_kernel_version	*Get_host_version*
host_processor_sets	*Pset_names*
mach_host_self	*Get_host_name*
processor_set_default	*Get_default_pset_name*

7.32 Requirements on *client* to *memory* Implementation Accesses

Request	Required Permission
memory_object_get_attributes	*Get_attributes*
memory_object_lock_request	*Invoke_lock_request*

7.33 Requirements on *client* to *proc* Implementation Accesses

Request	Required Permission
processor_control	*May_control_processor*
processor_get_assignment	*Get_processor_assignment*
processor_info	*Get_processor_info*
processor_start	*May_control_processor*

7.34 Requirements on *client* to *ps_name_port* Implementation Accesses

Request	Required Permission
processor_set_info	*Get_pset_info*

7.35 Requirements on *client* to *ps_control_port* Implementation Accesses

Request	Required Permission
processor_set_tasks	*Observe_pset_processes*
processor_set_threads	*Observe_pset_processes*

7.36 Requirements on *client* to *task* Implementation Accesses

Request	Required Permission
mach_port_extract_right	*Extract_right*
mach_port_get_receive_status	*Observe_pns_info*
mach_port_get_refs	*Observe_pns_info*
mach_port_get_set_status	*Observe_pns_info*
mach_port_names	*Observe_pns_info*
mach_ports_lookup	*Lookup_ports*
mach_port_type	*Observe_pns_info*
mach_port_type_secure	*Observe_pns_info*
mach_task_self	*Get_task_kernel_port*
task_get_assignment	*Get_task_assignment*
task_get_bootstrap_port	*Get_task_boot_port*
task_get_emulation_vector	*Get_emulation*
task_get_exception_port	*Get_task_exception_port*
task_get_kernel_port	*Get_task_kernel_port*
task_get_sampled_pcs	*Sample_task*
task_info	*Get_task_info*
task_ras_control	*Set_ras*
task_threads	*Get_task_threads*
vm_copy	*Copy_vm*
vm_machine_attribute	*Access_machine_attribute*
vm_read	*Read_vm_region*
vm_region	*Get_vm_region_info*
vm_region_secure	*Get_vm_region_info*
vm_statistics	*Get_vm_statistics*

7.37 Requirements on *client* to *thread* Implementation Accesses

Request	Required Permission
mach_thread_self	*Get_thread_kernel_port*
swtch	*Can_swtch*
swtch_pri	*Can_swtch_pri*
thread_abort	*Abort_thread*
thread_get_assignment	*Get_thread_assignment*
thread_get_exception_port	*Get_thread_exception_port*
thread_get_kernel_port	*Get_thread_kernel_port*
thread_get_sampled_pcs	*Sample_thread*
thread_get_state	*Get_thread_state*
thread_info	*Get_thread_info*
thread_set_state	*Set_thread_state*
thread_set_state_secure	*Set_thread_state*
thread_switch	*Switch_thread*

Section 8
Generic Security Server Requirements

This section describes the data structures and security requirements on security servers in general. The material in this section is applicable to all DTOS security servers that define policy for the DTOS kernel. This includes those security servers that define policy on exclusively kernel entities and those that define policy on higher level entities as well as kernel entities. Note, however, that security servers that define policy on only entities above the kernel level are not considered here.

Editorial Note:
The interface between the kernel and the security server described in this section is incomplete. Notification vectors have been omitted.

Each DTOS security server makes security policy decisions on a *subject security context* (SSC) to *object security context* (OSC) basis. For example, a security server implementing an MLS policy might define subject and object security contexts to consist of a single level. If the MLS policy required limiting subjects to operate in ranges of levels, then another alternative for the subject security context would be a pair of levels specifying the minimum level at which the subject may write and the maximum level at which the subject may read.

In the following, we use SSC and OSC to denote, respectively, the sets of subject and object security contexts. Note that these sets can be different for each security server.

Each security server recognizes certain sets of SSCs and OSCs. We use $\underline{r}ecognized\_sscs$ and $\underline{r}ecognized\_oscs$ to denote the SSCs and OSCs recognized by a given security server.

To hide the structure of security contexts from other system components, security servers use *security identifiers* (SIDs) to represent security contexts. Each security server recognizes certain sets of *subject SIDs* (SSIs) and *object SIDs* (OSIs). We use $\underline{r}ecognized\_ssis$ and $\underline{r}ecognized\_osis$ to denote the SSIs and OSIs recognized by a given security server. The OSIs $Task\_self\_sid$ and $Thread\_self\_sid$ defined in the formalization of the microkernel requirements are two special OSIs that must be recognized by all security servers. The kernel specifies one of these SIDs as a target to indicate that the security server should compute access for the client to the client itself.

Each security server processes access queries containing an SSI-OSI pair by mapping the pair to an SSC-OSC pair and performing an access computation. We use:

- $\underline{s}id\_ssc(ssi)$ to denote the SSC associated with ssi. This function is defined only for recognized SSIs.
- $\underline{s}id\_osc(osi)$ to denote the OSC associated with osi. This function is defined only for OSI other than $Task\_self\_sid$ and $Thread\_self\_sid$.
- $\underline{t}ask\_self\_osc(ssi)$ to denote the OSC representing that a task with SID ssi is accessing itself. This function is defined only for recognized SSIs.
- $\underline{t}hread\_self\_osc(ssi)$ to denote the OSC representing that a thread with SID ssi is accessing another thread within the same task. This function is defined only for recognized SSIs.
- $target\_osc(ssi, osi)$ to denote the OSC that is to be used for computing ssi-osi access. This function is defined only for recognized SSIs and OSIs. This function is defined as follows:

- If osi is $Task\_self\_sid$, then the result is $\underline{t}ask\_self\_osc(ssi)$.
- If osi is $Thread\_self\_sid$, then the result is $\underline{t}hread\_self\_osc(ssi)$.
- If osi is neither $Task\_self\_sid$ nor $Thread\_self\_sid$, then the result is $\underline{s}id\_osc(osi)$.

Each security server distinguishes between OSCs for communication ports, tasks, default pagers, pagers, threads, host name ports, host control ports, processors, processor set name ports, processor set control ports, kernel reply ports, and device ports. We use the following sets to denote the recognized OSCs for each class: $\underline{c}ommunication\_oscs$, $\underline{t}ask\_oscs$, $\underline{d}efault\_pager\_oscs$, $\underline{p}ager\_oscs$, $\underline{t}hread\_oscs$, $\underline{h}ost\_name\_oscs$, $\underline{h}ost\_control\_oscs$, $\underline{p}rocessor\_oscs$, $\underline{p}rocset\_name\_oscs$, $\underline{p}rocset\_control\_oscs$, $\underline{k}ernel\_reply\_oscs$, and $\underline{d}evice\_oscs$. These OSC classes address the Mach kernel entities. Security servers may also protect user level resources such as files. For example, a security server might define a set of OSCs called *file_oscs* for use with files. We use $\underline{r}ecognized\_osc\_classes$ to denote the collection of OSC classes recognized by a given security server. We require that $\underline{r}ecognized\_osc\_classes$ contains the class associated with each kernel entity and that each OSC in a recognized class is an OSC recognized by the security server.

Each security server associates a set of permissions with each OSC class. This set of permissions consists of those that are relevant to the type of entity represented by the class. For example, the set of permissions associated with $\underline{t}hread\_oscs$ must be $Thread\_permissions$. We use $\underline{o}sc\_class\_permissions(osc\_class)$ to denote the permissions a security server associates with osc_class. We require that the appropriate kernel permissions are associated with each kernel OSC class.

Each security server has an associated rule indicating which permissions are permitted on a context-to-context basis. We use $\underline{p}olicy\_allows(ssc, osc)$ to denote the set of permissions that a subject with context ssc is permitted to an object with context osc. Each permission set consists of a set of IPC permissions and a set of permissions specific to osc's class. We require that the latter set of permissions be contained in the set of permissions identified by $\underline{o}sc\_class\_permissions$. For example, when osc is the context of a thread port, the permissions returned by $\underline{p}olicy\_allows$ consist of IPC and thread permissions. Typically, a security server will manage a database that defines $\underline{p}olicy\_allows$. For example, a security server supporting an MLS policy will manage a database that defines the existing security levels and a partial ordering of those levels. We use $\underline{p}olicy\_database$ to denote this database.

Each security server has an associated rule indicating which permission decisions are cacheable on a context-to-context basis. We use $\underline{c}acheable(ssc, osc)$ to denote the set of permission decisions that are cacheable for a subject with context ssc to an object with context osc. Each permission decision set consists of a set of IPC permissions and a set of permissions specific to osc's class. We require that the latter set of permissions be contained in the set of permissions identified by $\underline{o}sc\_class\_permissions$. For example, when osc is the context of a thread port, the permissions returned by $\underline{c}acheable$ consist of IPC and thread permissions. Typically, a security server will manage a database that defines $\underline{c}acheable$. We use $\underline{c}acheability\_database$ to denote this database.

Each security server has an associated rule indicating the period of validity of access computations on a context-to-context basis. We use $\underline{v}alidity\_duration(ssc, osc)$ to denote the period of time for which an access computation on ssc to osc is valid. Typically, a security server will manage a database that defines $\underline{v}alidity\_duration$. We use $\underline{d}uration\_database$ to denote this database.

In summary, a generic security server consists of:

- sets of recognized SSCs, OSCs, SSIs, and OSIs,
- mappings from SSIs to SSCs and OSIs to OSCs,

- a partitioning of the recognized OSCs into recognized OSC classes,
- a mapping from recognized OSC classes to associated permission sets,
- a policy database and rule defining permission sets on an SSI-to-OSI basis,
- a cacheability database and rule defining sets of cacheable permission decisions on an SSI-to-OSI basis, and
- a duration database and rule defining validity durations on an SSI-to-OSI basis.

Each security server accepts requests specifying a pair of SIDs and a reply port. Each request is received through a message and each message has an attached sending ssi. Thus, a request can be viewed as a record having fields $client\_ssi$, $source\_ssi$, $target\_osi$, and ar_reply_name. In response to an access request, a security server sends a $Sec\_access\_provided\_id$ message containing the pair of SIDs, set of permissions, cacheability information (denoted by $control\_vector$) and validity duration. The set of permissions, cacheability and validity duration must be consistent with the pair of SIDs. If either of $source\_ssi$ or $target\_osi$ are unrecognized SIDs, then an empty set of permissions and duration of 0 must be returned. We allow the possibility of a non-empty set of cacheable permission decisions being returned. This provides the security server with a mechanism to avoid additional requests with the same pair of SIDs. Given a current security server state of ss_state, we use $Valid\_ss\_responses(ss\_state)$ to denote the set of messages which are consistent with the policy, cacheability and duration databases contained in ss_state.

The specification of a specific security server requires defining:

- The structure of SSC, OSC, $POLICY\_DB$, $CACHE\_DB$, and DUR_DB.
- The rule for defining $\underline{p}olicy\_allows$.
- The rule for defining $\underline{c}acheable$.
- The rule for defining $\underline{v}alidity\_duration$.
- The definition of any OSC classes beyond those representing kernel entities.
- The permissions associated with each OSC class beyond those representing kernel entities.

Section 9 Notes

9.1 Acronyms

CMU Carnegie Mellon University

DTOS Distributed Trusted Operating System

FSPM Formal Security Policy Model

IBAC Identity Based Access Control

IPC Interprocess Communication

KID Kernel Interface Document

MLS Multi-Level Secure

OSC Object Security Context

OSF Open Software Foundation

OSI Object Security Identifier

SID Security Identifier

SSC Subject Security Context

SSI Subject Security Identifier

VM Virtual Memory

9.2 Glossary

abstract service An abstract service is characterized by a relation on pairs of system states that specifies a change to a kernel data structure. For example, the service that creates a new task is characterized by a relation that specifies that the new system state contains a task that was not present in the old system state.

control point A control point is a point in the processing of a request where the kernel must enforce an access decision.

dirty page A page in kernel memory is dirty if the pager associated with the page has not yet been made aware of modifications that have been made to the page.

IBAC Server An IBAC server is a user space task that defines an IBAC policy on a memory object. The kernel interacts with an IBAC server in much the same manner as it interacts with security servers.

implementation service An implementation service is a Mach request for which the set of provided abstract services is difficult to formally define.

permission A permission is an access mode enforced by the kernel. The kernel ensures that a service is provided only when the client of the service has the appropriate permission.

precious page A page in kernel memory is precious if the pager associated with the page has indicated that it is not maintaining a copy of the page. Regardless of whether the page is dirty, the kernel must send the contents of the page to the pager before removing the page from memory.

security server A security server is a user space task that provides access computations to the kernel.

9.3 Open Issues

- This document does not currently address i386 and debugger requests.
- The original system design specified that each memory object would be labeled with IBAC protections and have an associated port used by the kernel to request IBAC decisions for the memory. These features have not been implemented and no policy requirements are stated regarding the IBAC protections and IBAC ports. However, permissions have been defined in the access vectors to control the use of these IBAC components should they ever be implemented.
- The specification of the interface between the kernel and the security server is not complete. In particular, requests to the security server include the permission being requested, and responses from the security server include a notification vector.
- Side effects need to be taken into account in some of the service definitions. This has been done for the thread services but might still need to be done for some of the other services.
- The prototype does not currently implement the enforcement of read-only access. The low-level memory routines in the prototype treat read and execute interchangeably.
- We need to consider whether the 12 permissions currently defined for memory control services can actually be reduced to a single permission indicating that the subject can serve as the pager for a given memory object. The case for doing this is that any usable pager probably needs to be allowed to use the entire paging protocol. Thus, the ability to page for a memory object may well be an all-or-nothing proposition. If so, nothing is gained by having 12 permissions.
- Checks on SID triples rather than SID pairs might be required to obtain the degree of control that we would like over some services. For example, checking permissions on the basis of the client, port, and receiver might be necessary when transferring a port right instead of checking on the basis of only the client and the port.
- The current prototype does not provide support for identity based policies in which each task and memory object might need a different SID. Enhancements to support such policies are under consideration. Few if any changes would be required in this documents to address such an enhancement.
- This control policy does not require complete tranquility of SIDs (the AID of a task, task port, or thread port may change) or tranquility of the set of accesses permitted between two SIDs. This may lead to inconsistencies between the prototype and this policy, because of possible concurrency in the system. In general, the requirements in this document state that a permission check is based upon the SID of the source and target in the state when the service is provided, though this cannot always be guaranteed in the prototype. One way to lessen the scope of this problem with respect to nontranquility of SIDs is to limit those permissions which can depend upon the AID field of the SID. This will likely be addressed in future drafts of this document. It is currently expected that the prototype will only use the AID fields for determining permissions to perform the services *ChangesTaskAid* and *CreatesTaskSecure*.

Appendix A Bibliography

[1] William R. Bevier and Lawrence M. Smith. A Mathematical Model of the Mach Kernel: Entities and Relations (Draft). Technical report, Computational Logic, Incorporated, April 1993.

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

[3] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[4] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, final draft edition, May 1993.

[5] NCSC. Trusted Computer Systems Evaluation Criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[6] Secure Computing Corporation. DTOS Formal Top-Level Specification (FTLS). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1994. DTOS CDRL A005.

[7] Secure Computing Corporation. DTOS Kernel and Security Server Software Design Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1994. DTOS CDRL A002.

[8] Secure Computing Corporation. DTOS Formal Security Policy Model (FSPM). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1995. DTOS CDRL A004.

[9] Secure Computing Corporation. DTOS Generalized Security Policy Specification. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1995. DTOS CDRL A019.

[10] Secure Computing Corporation. DTOS Kernel Interfaces Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1995. DTOS CDRL A003.

Appendix B
Prototype Security Server Requirements

This section describes the data structures and security requirements relevant to the prototype security server. This security server enforces accesses using both a Multilevel Secure (MLS) Policy and a Type Enforcement Policy. The MLS Policy provides confidentiality in the DoD sense. The Type Enforcement Policy provides a mechanism for constructing protected subsystems.

B.1 Security Contexts

Each *user* of the system is represented by a user identifier. We use $USER$ to denote the set of all user identifiers. Each subject and object has an associated *level*. We use $LEVEL$ to denote the set of all levels. Each subject operates in some *domain*. We use $DOMAIN$ to denote the set of all domains. Each object has an associated type. We use $TYPE$ to denote the set of all types.

The subject security context of a process consists of the following:

- $user$ — the user in whose name the process is executing,
- lvl — the level at which the process is executing,
- $domain$ — the domain in which the process is executing.

The object security context of an object consists of the following:

- $user$ — the user associated with the object. This is $null\_user$ except for those OSCs that are derived from an SSC (e.g., the OSCs in $\underline{t}ask\_oscs$).
- lvl — the object's level,
- $type$ — the object's type.

B.2 Policy Database

There are six components of the policy database. First, the policy database records the set of recognized security attributes. We use $\underline{r}ecognized\_users$, $\underline{r}ecognized\_levels$, $\underline{r}ecognized\_domains$, and $\underline{r}ecognized\_types$ to denote the sets of recognized security attributes.

Second, the policy database records the ordering of the recognized levels. We use $lvl_1 \preceq lvl_2$ to denote that lvl_1 and lvl_2 are recognized levels and that lvl_1 is at or below lvl_2. We require that $\preceq$ is a partial ordering of the levels.

Third, the policy database records a group of four permission sets for each domain-type pair. We use $\underline{t}e\_vectors(domain, type)$ to denote the group of permission sets associated with the pair $(domain, type)$. We view each group of permission sets as a record having the following fields:

- $same$ — the permissions for the case in which the source subject and target object are at the same level,

- $source\_higher$ — the permissions for the case in which the source subject is at a level strictly dominating that of the target object,
- $target\_higher$ — the permissions for the case in which the target object is at a level strictly dominating that of the source subject,
- $incomparable$ — the permissions for the case in which the levels of the source subject and target object are incomparable

We require that no permissions be allowed if the domain or type is unrecognized.

Fourth, the policy database records the set of levels for which each user is authorized. We use $\underline{a}uthorized\_levels(user)$ to denote the set of levels for which $user$ is authorized and require that unrecognized users are not cleared to any levels and recognized users are cleared to only recognized levels.

Fifth, the policy database records the set of domains for which each user is authorized. We use $\underline{a}uthorized\_domains(user)$ to denote the set of domains for which $user$ is authorized and require that unrecognized users are not authorized for any domains and recognized users are authorized for only recognized domains.

Finally, for each OSC class the policy database records a set of permissions that are AID-relevant. We use $\underline{a}id\_relevant(osc\_class)$ to denote this set. We will require that osc_class be an element of $\underline{r}ecognized\_osc\_classes$. In addition, the policy database records a set of domains $\underline{m}ay\_change\_user$ such that a context with a domain in the set may be granted AID-relevant permissions to contexts with a different user. A context with a domain not in this set will be granted an AID-relevant permission only to a context with the same user.

Together, these six components comprise the prototype security server policy database.

B.3 Cacheability Database

In the prototype Security Server there is no cacheability database since every permission sent in an access vector, whether granted or denied, is cacheable.

B.4 Duration Database

The duration database has the same structure as the type enforcement component of the policy database. For each domain-type pair, there is a separate duration value associated with each of the possible relations between the source and target levels. We require that non-zero durations only be permitted for recognized domains and types.

B.5 Prototype Security Server State

The prototype security server state is the generic security server state instantiated with the types $PrototypeSSC$, $PrototypeOSC$, $PrototypePolicyDatabase$, $NULL\_DATABASE$ and $PrototypeDurationDatabase$. The set of recognized SSCs is defined to be those for which the user field is a recognized user and the level and domain fields are appropriate for the user field. The set of recognized OSCs is defined to be those for which the user field is a recognized user, the level field is appropriate for the user field, and the type field is recognized. The functions $\underline{p}olicy\_allows$, and $\underline{v}alidity\_duration$ are defined in terms of the policy and duration databases. Both use $\preceq$ to determine the relation between the source and target levels and then return the

appropriate value of the four values associated with the source domain and target type. The function $\underline{cacheable}$ always returns the full set of possible permissions for the given OSC class.

The policy database is indexed by domain and type. For AID-relevant permissions the user fields of the contexts must also be considered when making the permission decision. For these permissions the policy database may be overridden. For example, in order for a task $task_1$ to create a task $task_2$ in a context with a different user, $task_1$ must be in a domain that is an element of the set of privileged domains $\underline{may\_change\_user}$. If the contexts of $task_1$ and $task_2$ have the same user, then permission is based entirely upon the domain and type.

The set of AID-relevant permissions is defined for each recognized OSC class. At a minimum the permissions $Cross\_context\_create$, $Change\_sid$, $Make\_sid$ and $Transition\_sid$ are AID-relevant for task OSCs.

Editorial Note:
We need to consider whether we want the prototype security policy to control which clients may check each task's permissions to each target.

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols

D

DTOS Services:

G

M

S