Coyotos Microkernel Specification

Version 0.6+

Jonathan S. Shapiro, Ph.D., Jonathan W. Adams
The EROS Group, LLC

September 10, 2007

Contents

Acknowledgments

Many people have assisted us in evaluating and advancing this design:

Norm Hardy, Charlie Landau, and Bill Frantz of the KeyKOS project. Charlie also runs the CapROS project, another successor to the EROS system.

The members of the `coyotos-dev` mailing list, notably Bas Wijnen and Tom Bachmann, Christopher Nelson, Dominique Quatravaux, and Pierre Thierry.

The members of the L4 community, notably Hermann Härtig, Espen Skoglund, and Kevin Elphinstone.

The members of the Systems Research Laboratory at Johns Hopkins University, notably Eric Northup, Swaroop Sridhar, and M. Scott Doerrie.

The external participants in the kernel design review meeting of 28-29 March, 2007: Godfrey Vassallo, John Davidsen, Scott Doerrie, and Norman Hardy.

There are surely others that we will come to name as the design stabilizes further, and some that we will inadvertently omit. To the last, please accept our apologies. As is customary, any flaw remaining in this specification is ours.

Comments and suggestions concerning this specification are welcome. They should be sent to the `coyotos-dev` electronic mailing list. In order to send, you must be subscribed to the list. The subscription interface may be found at:

`http://www.coyotos.org/mailman/listinfo/coyotos-dev`.

In order to keep the mail archives readable, we ask that you send only “plain text” emails.

Preface

Coyotos is a security microkernel. It is a microkernel in the sense that it is a minimal protected platform on which a complete operating system can be constructed. It is a security microkernel in the sense that it is a minimal protected platform on which higher level security policies can be constructed.

The Original Plan

As originally conceived, Coyotos was intended to be a relatively minor departure from its predecessor, EROS [6]. EROS [4] was a small, robust microkernel whose central design ideas were pervasive use of capabilities [11] as the fundamental access model, an atomic, blocking capability invocation (therefore atomic and blocking IPC) model , and a persistent single-level store [5]. All of these features were inherited with some revision from the KeyKOS system. [2] Early application-level work on EROS, notably the defensible network system [10] and the secure window system [8] revealed areas where the EROS architecture would clearly benefit from refinement, but did not initially suggest fundamental shortcomings in the architecture. Coyotos was to have been that minor refinement, incorporating a new IPC primitive called "endpoints" and a revised memory mapping entity called a PATT. Our main goals were cleanup, consistency, and formalization.

For algorithmic reasons, the PATT idea did not survive into the current specification, and has been replaced by guarded page tables [3, 1]. Though they were independently invented, guarded page tables may be seen as a generalization of the level skipping techniques of the KeyKOS translation mechanism or the Motorola MC68851 memory management unit [20]. The variant of guarded page tables incorporated here are modified to incorporate the fault handler and background space mechanisms of KeyKOS and EROS.

In January 2004, a summit meeting of sorts occurred between the several research groups working on L4 derivatives and Shapiro. The L4 Dresden group, in particular, wanted to get a better understanding of capability-based design and kernel mechanisms, with the intent that these would be adapted into the L4 architecture [9]. The new kernel architecture would come to be known as "L4.sec". There was some discussion of merging the two kernels, but no agreement could be reached on the future of L4's `map` and `unmap` operation. While the failure to merge the architectures was a disappointment, the idea that there would be a controlled experiment that would allow us to directly evaluate the `map/unmap` approach against the EROS `node` approach was a promising result in its own right.

Overrun by the Hurd

Events intervened in the form of Neal Walfield and Marcus Brinkmann, the current architects of the GNU Hurd system. The Hurd is a protected, object-based operating system that was initially constructed on top of the Mach microkernel. Mach has a variety of problems that have been thoroughly documented in the research literature. Of particular importance to Hurd are a lack of resource accounting mechanisms and poor performance. As a result of these issues, the Hurd project had provisionally decided to move to L4.

Unfortunately, modeling copyable, protected object references using L4's `map/grant` operations proved unexpectedly challenging. This left the Hurd project temporarily disrupted, leading Brinkmann and Walfield to seek more information about capability-based design. An extended discussion between Shapiro, Walfield, and Brinkmann at the *2005 Libre Software Meeting* about capability systems in general and the plans for Coyotos ensued. As more informa-

tion about the L4.sec design emerged [19], it became clear that copyable protected references might be problematic on the L4.sec interface as well. Walfield and Brinkmann traveled to Baltimore for a month-long set of design discussions in January 2006, leading to the current design for Coyotos.

In response to those discussions, we flirted for a while with introducing scheduler activations and a new IPC model. It didn't pan out. Initially, we thought that activations might be lighter weight than synchronous IPC. They aren't, and they introduce a lot of complexity in the exception handling model. Through editing errors, you may still find traces of that effort remaining in this document. If so, they are errors, and we would appreciate it if you might bring them to our attention.

Coyotos Today

The version of Coyotos described here has come full circle, and returns to the basic model of the EROS system. The primary differences are the introduction of endpoints, a first-class process object, and GPTs. It also reflects the January 2006 discussions between Walfield, Brinkmann, and Shapiro. As a result of those discussions, the architecture has been challenged a bit harder than it had been. Coyotos retains the atomicity and pure capability-based design of the EROS system.

Jonathan S. Shapiro, Ph.D.
The EROS Group, LLC
August, 2007

Chapter 1

Overview

This document describes the abstractions, objects, and interface specifications (capability types) implemented by the Coyotos microkernel. At some points it includes discussion of the intended model of usage by way of motivating or explaining what has been incorporated. Such discussions are non-normative.

All kernel-implemented objects are named and manipulated by means of capabilities, which grant varying degrees of authority according to the capability type. Developers can extend the system with new objects by deploying processes that implement the associated interfaces. Several such application-implemented objects are part of the core Coyotos system.

1.1 Microkernel Objects

The Coyotos kernel provides processes, GPTs (mapping structures), schedules, receive queues, pages, and a small number of other kernel objects.

Processes Processes are the unit of execution, scheduling, and resource binding. A process names its address space, its schedule (which governs their execution timing) and their fault handler (which receives notice of exceptions).

Schedules Schedules are an abstraction of computational resources. In order to execute instructions, a process must name (via a capability) the schedule under which it runs. The schedule, in turn, must convey authority to use one or more processors under a defined scheduling contract.

GPTs GPTs are the unit of address mapping composition. An address mapping is defined as a mapping from addresses to capability slots, and is represented by a directed (potentially cyclic) graph of GPTs whose leaf capability slots name atomic storage units (pages or capability pages). A virtual address is divided into a **virtual page address** and a **page offset**. Valid virtual page addresses describe paths to leaf slots that contain data page or capability page capabilities.

Endpoints An endpoint is a named rendezvous point between a message sender and a message receiver. Each endpoint carries a receiver-interpreted endpoint identifier. In addition, each endpoint provides means for ensuring that its capabilities can be used exactly once.

Receive Queues Receive queues provide a means for several processes to receive from a single endpoint. The receive queue acts as a rendezvous point for the receiving processes. When a message is sent via the endpoint, the kernel will select a waiting process from the receive queue and deliver the message to that receiver. This permits kernel demultiplexing of receive processes, which enhances performance on multiprocessors.

Receive queues remain an experimental idea, and are not implemented by the current kernel.

Pages Pages are the atomic unit of data and capability storage allocation. An address space consists of a lattice of GPTs whose leaves are pages. Pages are typed: a page may contain either data or capabilities, but not both. The size of a page is determined by the underlying hardware architecture.

There are a small number of other kernel-implemented capabilities. These primarily provide protected transformation

operations on capabilities.

1.2 Entry Capabilities and Extensibility

Endpoint objects have “entry capabilities”. An entry capability does not implement operations on the endpoint. Instead, it provides the means by which an application introduces new services. Any invocation of an entry capability is delivered to the providing object server.

1.3 Checkpointing and Persistence

Coyotos is a persistent object system. Main memory is treated as a *cache* of a larger backing store. Objects are loaded from backing store on demand and are rewritten to the backing store as a consequence of age or checkpoint. Following a system restart, persistent objects retain their state as of the last checkpoint. A checkpoint saves a “consistent cut” of the system. In consequence, processes are recovered in such a way that ongoing communications on the local machine may be resumed without recovery effort.

Secure Restart On restart, any connection to the outside world is *severed* if continued communication on that connection might (conservatively) require re-authentication. In particular, network and terminal connections are terminated.

Lost Objects One risk in this class of design is that objects may be permanently lost as a consequence of low-level storage failures (e.g. sector errors). When backing store is not already duplexed, the Coyotos object store implementation uses software duplexing of critical system structures. Applications may also use this mechanism if desired.

The checkpoint management interfaces used in a driverless kernel are still being refined, and are not yet included in this specification.

1.4 Process States and Exceptions

From the kernel perspective, a process has five run states: **blocked**, **faulted**, **receiving**, **ready**, and **running**. A blocked process is waiting for a kernel resource. A ready process is attempting to execute instructions and is waiting for a CPU. A running process is currently executing. A faulted process is not attempting to initiate instructions.

When a process incurs an exception, the Coyotos kernel synthesizes a message on behalf of the faulted process to a fault handler. It is the responsibility of the fault handler to decide what to do. The kernel does not define a fault handling policy.

1.5 Messages

From the sender perspective, message transmission is (nearly) atomic. From the receiver perspective, message transfer occurs asynchronously. Arrival is signalled by a message completion event delivered to the receiver’s activation handler.

Relaxed Data Atomicity Coyotos permits relaxed data atomicity for stateful messages. While a stateful receive is pending, the data bytes of the receive area are considered undefined and may be modified by the kernel to arbitrary values. When receipt has completed, the receive area is defined up to the kernel-provided length of the received message. The relaxed atomicity rule allows the kernel message send implementation to avoid a pre-probe pass on the received data area, which significantly improves performance. Note that the ”undefined” rule explicitly does *not* apply to received capabilities. The complete set of capabilities (if any) transferred by a message are required to be transferred to receiver-controlled storage atomically. This requirement ensures that the inductive state transition requirements of the formal capability protection model are satisfied.

Blocking Send A blocking send guarantees eventual delivery provided the operation completes and the receiver is not destroyed before delivery. Page faults at the receiver's designated receive location(s) will be delivered to the receiver-designated fault handlers as required. When fault handling has completed, the sender will retry the send operation from the beginning.[1] Senders may implement watchdog timeouts on send operations by arranging to post exceptions to themselves after a timed delay.

Non-blocking Send A non-blocking send will be silently discarded if any condition arises that would cause a blocking send to block. It will be truncated if a receiver page fault occurs during transmission. If truncation occurs, the receiver is notified of the partial delivery.

1.6 Naming and Invocation

Coyotos objects are named by capabilities. A capability is a kernel-protected value that names a resource and identifies some interface (equivalently: facet or object) of that resource. The interface in turn defines methods that the invoker can invoke by sending a message specifying the corresponding method code point. Thus, every invocation consists of a message send to a particular method of a particular interface of a particular resource, performed by invoking a capability. This is true both for server-implemented interfaces and kernel-implemented interfaces.

The Coyotos invocation mechanism is derived in part from the EROS design. The invocation payload has been enriched, but the invocation state model has been simplified. An invocation consists of a send phase followed by an optional asynchronous receive phase. The send phase may specify blocking or non-blocking behavior. If a non-blocking send is unable to make immediate progress, its message payload is truncated or dropped. The receive phase, if present, blocks until an incoming message arrives, and can optionally require that the incoming message arrive on a particular endpoint identifier.

The Coyotos kernel implements only one major system call: `InvokeCap`. A small number of additional system calls exist to implement pseudo-instructions such as capability load and store.

Entry capabilities contain a 32-bit protected payload field. The endpoints that they name contain a 64-bit endpoint identifier. Both values are delivered to the recipient as part of an incoming message. Neither is readable or modifiable by the capability's invoker. Servers may use these values to distinguish interfaces, object identities, permissions, or other desired characteristic.

1.7 Exception and Interrupt Handling

For reasons of performance, the Coyotos kernel handles scheduling-related interrupts directly. It does not specify or implement a policy for other interrupt handling. The kernel maintains a capability-named interface for interrupt handler registry. With the exception of low-level scheduling preemption, all policy and processing associated with interrupts is handled by application-level code.

The Coyotos kernel also pushes responsibility for exception handling policy to application level. When runtime application exceptions occur, the kernel delivers the state associated with the exception to an external fault handler designated by an endpoint.

1.8 Protection Model

An essential part of the security microkernel concept is that security policy — including mandatory security policy — should be implemented by application code. The code that enforces system-wide policy needs to be protected and must not be evaded, but it does not necessarily need to run in supervisor mode.

In keeping with this philosophy, the Coyotos kernel does not implement a security policy. Coyotos provides primitive protection support in the form of protected capabilities. Applications can invoke services only by invoking capabilities.

[1] The need to retry from the beginning is onerous, and the specification will eventually be refined to allow optimization in this case.

Capabilities are kernel protected, and can be obtained only by transfer over capability-authorized channels. It has been shown formally that this restriction is sufficient to support (overt) confinement of subsystems [7], and that given overt confinement, a higher-level security policy can be implemented either by construction or by an application-level reference monitor [18].

A useful property of capability systems is that they directly express the "relies-on" relationships between components. If an object or subsystem *A* depends directly on a second object or subsystm *B* for its operation, then *A* necessarily holds a capability to *B*. In the absence of such a capability, *A* cannot invoke *B* at all (or even know if the existence of *B*). A key point here is that *A* may rely on *B* only in a qualified way, and (in some cases) may be able to take measures to guard against failures or hostility from *B*. This allows applications to take direct responsibility for their dependencies, and also to impose context-sensitive access restrictions on their providers.

For this reason, we try to avoid the term "trust" in our designs, preferring instead to use "relies on."

Part I

Microkernel Abstractions

Chapter 2

Capabilities

The Coyotos kernel implements a number of object types, each of which has a corresponding capability type:

Encoding	Type	Description	Restrictions
0	Null	Universal, invalid capability.	
1	Window	A local mapping window (Chapter 4).	RO,NX,WK
2	Background	A background mapping window (Chapter 4).	RO,NX,WK
3	KeyBits	Discloses the bit representation of capabilities.	
4	Discrim	Classifies capabilities.	
5	Range	Fabricates object capabilities.	
6	Sleep	Interface to the kernel interval timer.	
7	IRQ Control	Interrupt request line control interface.	
8	Schedule Control	Interface to the kernel master scheduling table.	
9	Checkpoint	Control capability for the kernel checkpoint mechanism.	
10	ObStore	Interface between kernel and object store manager.	
11	Pin Control	Permission to pin objects in memory.	
12	Schedule	Permission to execute under a particular schedule.	
13	SysCtl	Start, stop system, enter sleep states.	
14	KernLog	Append text to kernel log.	
15	IOPriv	Authority to read/write IO ports.	
16	IrqWait	Authority to wait for an arriving interrupt.	
17-31	*Reserved*	*Encodings reserved for future use.*	
32	Endpoint	Control capability for an endpoint.	
33	Page	Data page. In general, the size of a page is determined by the underlying hardware page size. Device pages may be any power of two larger than this.	RO,NX,WK
34	CapPage	Capability page. The size of a capability page is determined by the page size of the underlying hardware page size.	RO,NX,WK
35	GPT	Guarded Page Table. Used to compose larger address spaces from pages.	RO,NX,WK,OP
36	Process	Capability that manipulates the kernel process abstraction.	
37	AppNotice	Capability that permits posting of non-blocking, application-defined software notices.	
38-62	*Reserved*	*Encodings reserved for future use.*	
63	Entry	Authority to send to the process designated by an Endpoint.	

The **RO**, **NX**, **WK**, and **OP** restrictions respectively indicate, read-only, non-executable, weak, and opaque permission restrictions. These are described in detail in the chapter on address spaces.

2.1 Representation

A capability is 16 bytes, and uses the same representation on both 32-bit and 64-bit platforms. The capability structure is a "tagged union" whose details depend on the capability type field. The kernel is entitled to use optimized representations internally. The representation given below is the representation disclosed by KeyBits, which is the representation typically used on disk.

Except where otherwise indicated, reserved fields must be zero-filled. The **P** (prepared) bit and the **hz** (hazard) bit are kernel internal, and are always zeroed by `keybits` when the capability representation is returned.

2.1.1 Capabilities to Memory Objects

Memory capabilities include page, cappage, GPT, local window capabilities, and background window capabilities. All of these are used to describe portions of the address space. The format of page, cappage, and GPT capabilities is:

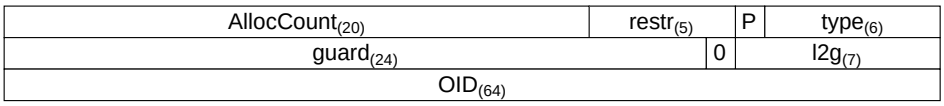

Figure 2.1: Memory object capability

The format of a window capability is:

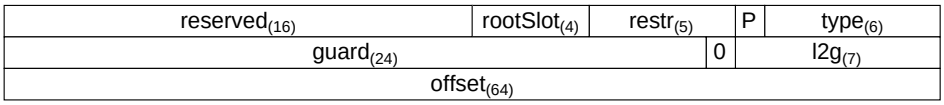

Figure 2.2: Mapping window capability

The *rootSlot* field of the window capability is used only for local window capabilities, this field is reserved in background window capabilities.

Invariant: `l2g` $\leq$ `64`
Invariant: `(l2g == 64)` $\Rightarrow$ `(guard == 0)`[1]
Invariant: `((guard` $<<$ `l2g)` $>>$ `l2g) == guard`[2]
Invariant: `l2g` $\geq$ `log2(`*`page size`*`)`
Invariant: `(offset mod` $2^{\texttt{l2g}}$`) == 0`

These invariants are ensured by the operations that fabricate the respective capabilities. The balance of the system is entitled to assume that they hold.

When traversing a memory capability, the virtual address `va` is defined as the bitwise concatenation of three fields `g`+`u`+`v`, where `g` is a variable length, possibly empty bit string that will be used as a guard value, `u` is a variable length, possibly empty bit string that will be used to index into the slots of the named GPT (if any), and `v` is the virtual address that will remain to be translated at the next step (if any). The length |`v`| is determined by the `l2v` field of the named Page, CapPage, or GPT. The capability field `l2g` contains length of the bit string |`u`+`v`|. The value of the effective guard is a multiple of $2^{\texttt{l2g}}$. For page and capability page capabilities, the value `l2g` also specifies the target page size. This is possible because neither pages nor capability pages have slots to be indexed.

[1] This invariant ensures that no bounds check is required before performing C shift operations whose size equals the machine word size. On most architectures such shift operations truncate the shift amount, but if the value being shifted is zero any shift (including none at all) will produce the right answer during address space traversal.

[2] This invariant ensures that any bits of the guard value that would appear (after the normalizing shift) above the highest valid bit position in an address must be zero.

2.1.2 Message-Related Capabilities

Endpoint capabilities currently do not carry permission bits, but are otherwise similar in layout to memory capabilities. The protected payload field is reserved in the respective control capabilities, and should be zero.

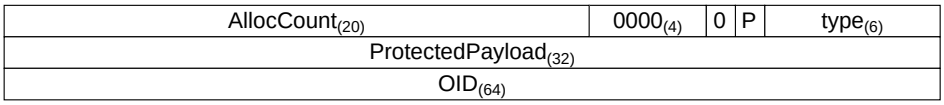

Figure 2.3: Endpoint capability

Invocations of an endpoint capability ignore the protected payload and provide access to the kernel-implemented object. Invocations of an Entry capability are delivered to an implementing process designated by the endpoint. The protected payload field of the endpoint capability is provided as an additional output of the invocation.

2.1.3 Capabilities to Processes

The format of a process capability is:

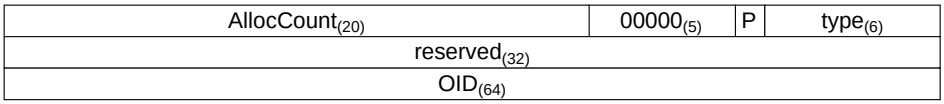

Figure 2.4: Process capability

2.1.4 Miscellaneous Capabilities

The format of a miscellaneous capability is:

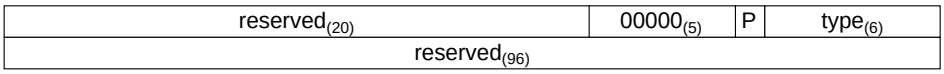

Figure 2.5: Miscellaneous capability

2.2 Valid Capabilities

Wherever this specification refers to a capability of a specific type, it should be taken to mean a *valid* capability of the stated type. The meaning of an invocation of a valid capability is determined by its implementation provider (kernel or server).

A non-object capability is any capability whose external representation does not include an allocation count. A non-object capability is always valid. Non-object capabilities are not revocable.

All other capabilities are object capabilities. An object capability is valid if and only if all of the following conditions are met:

- There exists some object with a matching object identifier (OID) whose type is compatible with the type of the capability.[3]

  This condition may be violated if backing store is lost or corrupted, or through a bug in the object manager.

[3] This specification intentionally does not take a position on whether OIDs are globally unique or only unique with respect to objects of compatible representation type. This choice is left to the implementation. However, the specification reserves the OID range [0xff00000000000000ull, 0xffffffffffffffffull] to refer to device memory page frames. See Part I and the specification of the Range capability for discussion of this.

- The allocation count in the capability matches the allocation count in the object.

  This condition ceases to be true when an object is revoked (see `coyotos.range.rescind`).

- In the case of an endpoint capability whose `PM` bit is set, the protected payload field of the endpoint capability matches the protected payload field of the endpoint object that it names.

All other object capabilities are invalid. An invalid capability behaves in all observable respects as if it were the Null capability. This applies both to invocation of an invalid capability and to operations that act on invalid capabilities (notably KeyBits, which has implications for debugging invalid capabilities). The kernel is free to overwrite any capability location with a Null capability when it determines that the capability contained in that location is invalid.

2.3 Capability Prepare

Coyotos is an object paging system. Both object load and object unload are driven by the use of capabilities. Ignoring latency, this paging behavior is normally invisible to applications. The exception is that object page-in may reveal low-level storage failures that make an object unrecoverable.

Whenever a capability is used, the kernel internally performs a *prepare* operation on the capability. Conceptually, this prepare step is being done by the process that is performing the current system call. The prepare operation may have several outcomes:

- If the capability is a non-object capability, the prepare operation succeeds (by definition).

- If the capability names an object, but its `allocCount` does not match the `allocCount` of the corresponding object, the capability is re-written (in place) to the Null capability.

  The containing object is not marked modified. If other operations cause the containing object to be modified, the Null capability will be written to disk. Otherwise, subsequent reloads of the object will re-obtain the stale capability and this check will be performed again with the same result.

  Several optimizations and mechanisms are used to ensure that the disk allocation count does not overflow.

- If the object named by the capability is not in memory, steps are taken to load it. The preparing process is enqueued to wait for the completion of this request, and re-starts its operation when the object has been loaded. In rare cases, this step may result in an `ObjectContentLost` exception if the backing store has experienced an unrecoverable storage error.

- If the object named by the capability is in memory, it is locked for the duration of the current system call unless they are unlocked explicitly.

A capability is “used” if:

- The capability is invoked by the current system call.

- The capability designates the invokee of the current system call.

- Fetching a capability argument or storing a capability result requires memory traversal, in which case all capabilities in the traversed slots are used.

- The operation requested by the current system call accesses or mutates the target object of the capability.

Coyotos implementations are required to be atomic. This implies that all resource acquisitions (and therefore all capability prepares) must be acquired before any observable side-effect of a system call occurs.

2.4 Extensibility

Coyotos is an extensible object system in the sense of Hydra [16]. New objects may be introduced by designing a process that implements the desired object. Capabilities to these objects are implemented as Entry capabilities. The kernel checks these capabilities for validity, and optionally for a protected payload match (see Chapter 5), but does not otherwise define semantics for these capabilities.

Because the kernel does not know the semantics of these extensions, entry capabilities are not considered "safe" by the `coyotos.discrim.classify` operation.

Chapter 3

Processes

A Coyotos process provides an abstraction of the user-mode execution engine presented by the underlying microprocessor. From the kernel perspective, a process is the unit that is dispatched by the kernel for execution.

Coyotos does not distinguish between processes and threads. A process encapsulates a single kernel thread of execution. Coyotos address spaces are first-class objects. Two (or more) processes may be constructed that designate the same address space. This achieves concurrent execution of multiple kernel threads of control within a common addressing environment and resource pool.

Coyotos implements the system calls described in Chapter 6. Most of these should be viewed as software-defined instructions. The exception is the `InvokeCap` system call, which performs capability invocation (see Chapter 5). The majority of the kernel's function is provided in the form of kernel-implemented objects (equivalently: services) that are named by capabilities. These services are invoked in the usual way by invoking their capabilities.

3.1 State of a Process

The state of a process may be divided conceptually into kernel (privileged or sensitive) state and user (non-privileged) state. **User state** is that state which a process may modify directly without kernel intervention. This includes architecture-defined non-privileged register state. It also includes additional "pseudo registers" defined by Coyotos that support the capability invocation mechanism.

Kernel state is that state which records or discloses protection information, or for which the kernel must guarantee invariants for reasons of security, robustness, or operational consistency. The representation of capabilities, for example, is kernel state. The Coyotos process structure contains space to save both the kernel state and the user state of a process.

On some hardware architectures, the separation between kernel state and user state is not cleanly accomplished by the architecture. The most common examples of this involve design failures in the architected processor status word. The IA-32 EFLAGS register, for example, includes state such as the supervisor mode bit and the current "IO privilege level." Such fields present a problem because the balance of the EFLAGS register must be modifiable by untrusted code. When an unprivileged application runs normal instructions, the hardware generally protects these bits from modification. On such architectures, Coyotos must ensure that any registers modified by get/set registers and similar operations properly protect these fields. The architecture-specific annex for each architecture identifies any such registers and their update constraints.

3.1.1 Per-process State

Each process has the following state:

- The process run state. This field indicates whether the process is running (0), receiving (1), or faulted (2).

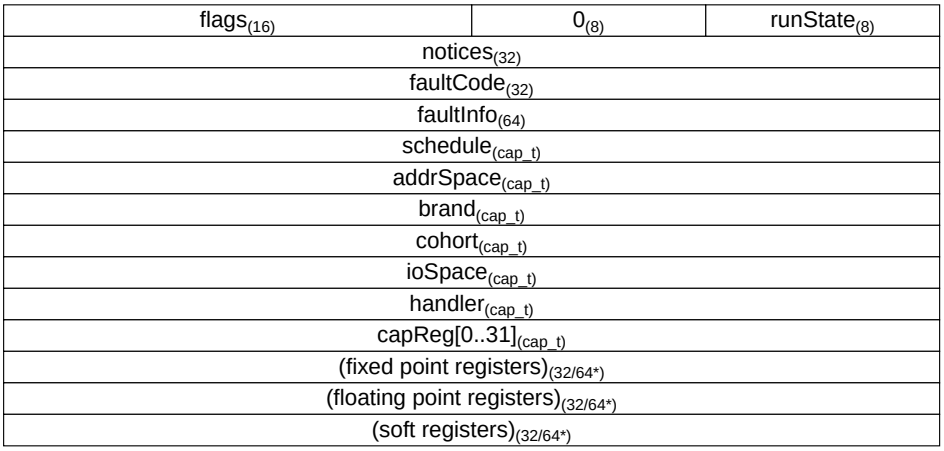

Figure 3.1: Per-Process kernel state

Figure 3.2: Process flags word

- The process flags word. This word contains the process run state and several bits that control fault-related and debugging-related behavior.
- A software-defined notices bitfield, **notices**, indicating (by bit position) the software-defined notices that are pending for this process.
- 32 capability "registers" that are implemented in software by the Coyotos kernel.
- Capability slots that support process recognition and identification: **brand** and the **cohort**.
- Capability slots that identify resources on which the process depends: the address space, the schedule, and the external fault handler.

  Coyotos address spaces are "first class". An address space may exist without having any associated process. Multiple processes may name the same address space by placing the same address space capability in their respective address space slots. Schedules are similarly "first class."
- Slots related to exception handling.

  The **faultCode** and **faultInfo** are conceptually similar to the underlying hardware processor's exception registers. A process-incurred exception causes these registers to be updated with the information necessary for error diagnosis and possible resolution. The exception fault code space unifies both hardware-defined and kernel-defined exceptions into a single code point space.

  The **handler** capability slot contains an entry capability to the external fault handler (if any). This is an external process that should be notified whenever this process incurs a fault.
- Storage for the architecture-defined non-privileged register set. Access to these registers is by means of invocations on the architecture-specific process capability.

The per-process capability state, fault code, and fault information can be accessed and manipulated only through invocations of the process capability.

The process flags are shown in Figure 3.2. The fields have the following meanings:

Field	Meaning
xm	**Execution Model** indicates whether this process uses a 32-bit (0) or 64-bit (1) execution model. This bit is significant mainly on architectures having multiple execution models, such as `amd64`. It controls certain aspects of cross-model invocations.
sx	**Slice Expired** This bit is set by the kernel when the process's real-time slice has expired. The slice expired event is considered an application-defined interrupt. Use and delivery of the **sx** notification is discussed in Chapter 7.
sn	**Soft Notice** This bit is set by the kernel whenever a new bit is set in the pending application-defined notices field.
tc	**Trap On Call** indicates that the process should incur a "trap on syscall exception when it attempts to perform a system call. This trap will occur *after* registers have been saved to the process structure, but *before* arguments have been examined by the kernel. In particular, the system call number will not yet have been examined by the kernel.
tr	**Trap On Return** indicates that the process should incur a "trap on system call return exception when it exits or bypasses the receive state following a successful invocation. This trap occurs just *after* the parameter words (if any) have been copied out to the application. In consequence, it occurs after any associated exceptions are processed by the recipient. Control has not been returned to the receiver. If this bit is set at process system call return, a `process.FC_SysCallReturn` fault will be set in the process state just prior to returning. The `process.resume()` does not cause the invokee to resume in the system call exit path, so this exception will not re-occur on resumption.
cs	**Call Step** if set (1), indicates that the **tc** bit should be ignored at the next point where it would normally take effect. This bit is set as a side effect of the `process.resume()` operation if the currently pending fault code is `process.FC_SysCallEntry`. It is cleared whenever the process proceeds successfully to the commit point of the current system call.
pc	**Parameter Copyout** This bit is set by the kernel whenever a parameter copyout from the parameter scratchpad area is required before resuming user-mode execution of the current process.

3.2 Execution Model

The instruction set available to a Coyotos process consists of the user mode (non-privileged) instruction set of the underlying processor architecture, the kernel-implemented `InvokeCap` instruction (which is the subject of Chapter 5).

From the perspective of the kernel, a process exists in one of the following run states:

blocked Process is attempting to send, but is blocked availability of a kernel resource. Process has no current or pending software interrupts. On release, process will resume in the *ready* state.

receiving Process is waiting for an incoming message from an endpoint, and has no current or pending software interrupts. On receipt, process will resume in the *ready* state.

ready Process is attempting to execute instructions. Process may have current or pending software interrupts. Pending exceptions will be delivered when the process transitions to the *running* state.

running Process is assigned to a CPU and is executing instructions.

faulted Process has incurred an exception that has been reported to the external fault handler designated by the process's `handler` capability. Process will not executing instructions.

The state transition diagram is shown in Figure 3.3.

The *blocked* state is not externally observable. A blocked process has an externally reported `runState` of "running". Such a process is deemed to be running without making progress or consuming CPU cycles. The receiving state it also not externally observable. A receiving process is executing the receive phase of a capability invocation very slowly.

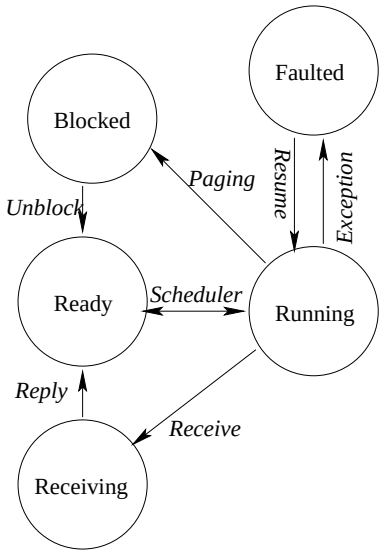

Figure 3.3: Process state transitions

A process that is in the *running* state will initiate instructions as long as its process `faultCode` field is set to `FC_NoFault` (0). Execution behavior when any other value is stored in the `faultCode` field is discussed in Section 3.3.

3.3 Exception Handling

An exception occurs as the result of an instruction executed by the process. Every exception has an associated fault code. Specific exceptions may define an additional pointer value to be delivered as additional fault information. The fault code and fault information are delivered to the process by storing them in the `faultCode` and `faultInfo` fields of the Process and causing the process to resume execution.

3.3.1 Exception Delivery

When a process attempts to initiate instructions with a `faultCode` other than `FC_NoFault`, the behavior is as follows:

1. If an Entry capability is stored in the process's `handler` slot, the kernel synthesizes a message to this endpoint on behalf of the faulting process. The message will provide the `faultCode` and `faultInfo` values and a process capability to the faulted process. Disposition of the faulted process is now at the discretion of the fault handler.

   If the handler process is blocked, handler message delivery will be re-attempted when the external handler process becomes unblocked.

2. The process enters the *faulted* state (runState = stopped) and ceases to execute instructions.

In the absence of a specified external handler, a process attempting to deliver a fault notification to its external handler will effectively cease to execute instructions without notice to anyone. It is the responsibility of the programmer to ensure that an external handler capability is defined if noticing this condition is required.

Note that the state of the per-process `handler` slot is checked on each delivery attempt. If a process blocks attempting to deliver its fault information to an external fault handler, and the `handler` slot is modified before the external handler becomes unblocked, the fault may end up being delivered to a different handler or to no handler at all depending on the new value of the `handler` slot.

3.3.2 Return From Out-of-Process Handler

If an exception has been delivered to a handler, the handler must take action to clear the fault. It does this by invoking the process capability provided by the kernel upcall to clear the fault and return the process to the running state.

3.4 Application-Defined Notifications

Coyotos supports application-defined non-blocking notifications via the `AppNotice` capability type. A notification is posted by invoking the `AppNotice` capability with 32-bit mask indicating the notifications (in the range 0..31) to be posted. The set of authorized notifications is determined at the time the `AppNotice` capability is fabricated. The effect of posting a set of application-defined notices is to set the corresponding bits of the target process `softNotices` word, and to set the `sn` bit of the target process flags if the value of `notices` has changed as a consequence of this posting (i.e. if the notice was not already pending). Of the set of notices posted, only the authorized subset is delivered.

If any notices are pending when the recipient enters an open wait, they will be delivered as a message, with a specified endpoint ID of `~0ull` and a protected payload value of zero. Delivery of pending notices has higher priority than other incoming messages.

Delivery of application-defined notices is suppressed during a closed wait.

Chapter 4

Address Spaces

The Coyotos architecture defines 64-bit address spaces for both 32-bit and 64-bit machines. On 32-bit machines, the leading 2^{32} byte positions are addressable by hardware load and store instructions. That is, the hardware-accessible map is a *window* onto the leading subrange of the software-defined space.

On some architectures, a portion of the hardware-addressable space may be reserved for use by the kernel. On such machines, the hardware-accessible address space is overlaid by the kernel-defined region.

4.1 Memory Objects and Address Interpretation

Three objects are used to define Coyotos address spaces: pages, cappages, and GPTs. Capabilities to these objects may be invoked in the usual way. The interface definitions for these objects are provided in Part II.

The meaning of a data (capability) address reference is determined by starting at the data (capability) address space capability of the referencing process and traversing memory objects until the address has been successfully translated or an exception has occurred. The traversal process is similar to the traversal of hardware-based hierarchical translation tables, but there are several differences:

- The Coyotos mapping structures provide support for per-region fault handlers. Any region of size 2^k pages may have an associated fault handler. When a memory fault is reported to the in-process fault handler, the in-process handler may optionally forward memory fault messages to the per-region handler in order to request region-specific fault handling.
- The "levels" of the mapping hierarchy are dynamically determined. Smaller subspaces may appear where a larger space is expected, with the effect that the "missing" regions are considered invalid addresses. Larger subspaces may appear where a smaller space would naturally appear, with the effect that only the leading subrange of the larger subspace is addressable through this mapping.
- A mechanism is provided for mapping "windows" onto other address spaces by reference. This enables one address space to map (portions of) another even when the second space is opaque.
- In order to support certain essential types of addressing flexibility — notably windows — it is necessary to allow some unusual arrangements of the hierarchical structures. An unfortunate consequence of this is that it is possible for a hostile or erroneous program to create statically cyclic address spaces. Such spaces are **malformed**, and attempts to reference a cyclically defined address generate a `MalformedSpace` exception.

4.1.1 Permissions

All memory object capabilities carry a field, "restr", which specifies restrictions on which types of access may legally be performed through that capability. The RO, NX, and WK bits may be set on any memory capability type. The OP

and NC bits are meaningful only for GPT objects. The CD and WT bits are meaningful only for device pages.

RO Read Only (0x1) Attempts to perform write references along any address translation path that traverses this capability are prohibited.

NX No Execute (0x2) Instruction fetch references along any address translation path that traverses this capability are prohibited. Attempts to perform instruction fetches at such addresses generate an `NoExecute` exception.

> **Issue**
>
> I have not yet examined the exception handling policy for machines that implement **NX** to confirm that a differentiated access violation type is generated at the hardware level.

On hardware that does not support the **NX** restriction, the **NX** bit is ignored.

WK Weak (0x4) A capability read reference along an address translation path that traverses a capability with this bit set conservatively downgrades the returned capability, if required, in a way that ensures *transitively* read-only authority.

Capability and data stores that traverse a weak capability in the translation path generate an access violation exception.

OP Opaque (0x8) The address space structure may not be accessed or modified through any GPT capability with the Opaque bit set. This bit is meaningful only for GPT capabilities.

NC No-Call (0x10) The address space structure may not be accessed or modified through any GPT capability with the Opaque bit set. This bit is meaningful only for GPT capabilities.

CD Cache Disable (0x8) Indicates that the content of this page must not be cached. Reads and writes must be issued to the main memory bus precisely in the order and reference size specified. This bit is meaningful only for page capabilities that name device pages.

WT Write Through (0x10) Indicates that writes to this page must be passed immediately and precisely to the main memory bus. This bit is meaningful only for page capabilities that name device pages.

The result of translation of the form `translate(space,addr,access-type)` is either an exception or a valid translation of the form `(page,offset)`. If an exception is generated, the type of the exception and the originally referenced address are reported to a handler (if one is defined), the faulting instruction (if any) has no effect), and the program counter is not advanced. The defined reference types are:

Fetch	Instruction load from address space.
Load Data	Read data from address space.
Load Capability	Read capability from address space.
Store Data	Write data to address space.
Store Capability	Write capability to address space.

4.1.2 References and Access Violations

The rules for address translation are given below in the discussions of individual memory objects. As traversal of the memory objects proceeds, the *effective restrictions* associated with the address are computed by beginning with no initial restrictions and performing a cumulating logical *or* with the restriction bits in each traversed capability as translation progresses.

If a capability is traversed during translation that cannot legally appear within an address space, a `Malformedpace` exception is generated according to the reference type.

If the traversed path is well-formed, but the address cannot be completely translated, an `InvalidAddress` exception is generated according to the reference type. Untranslatable fetch references generate the `InvalidAddress` exception.

If an address is completely translatable, the resulting permission restrictions may not permit the reference type. In this case an exception will be generated according to the following rules:

Ref Type	**Permissions**	**Result**
Fetch	NX	Exception: `NoExecute`
Capability Store, Data Store	RO *or* WK	Exception: `AccessViolation`

If the permissions are sufficient to allow the operation, a final check is made to ensure that the type of the load or store operation (data or capability) matches the type of the page mapped at that address (Page, CapPage). If a type mismatch occurs, a `DataAccessTypeError` or `CapAccessTypeError` exception is raised.

A capability load that traverses a path having `WK` restrictions will succeed, but will return a downgraded result as follows:

Capability At Address	**Result**
Page, CapPage, GPT, Window, Endpoint	Copy with RO, WK bits set.
Discrim	Return value is unchanged.
other	Null capability is returned.

4.2 Pages and Capability Pages

The smallest mappable unit, and therefore the smallest address space, is the page or the cappage. A page is the atomic unit of data storage whose size is implementation-defined. A capability page is a page-sized unit that holds capabilities rather than data. Capabilities are byte-addressed opaque 16-byte quantities that are aligned at 16 byte boundaries.

With the exception of device pages, Coyotos implements a single page size whose size matches some hardware page size implemented by the underlying hardware. This size is specified by the architecture-specific annex. On processors that implement multiple page sizes, the selected page size need not be the smallest size supported by the underlying hardware. It is implementation-dependent whether the kernel will attempt to exploit larger hardware page mapping sizes if available. If such exploitation is attempted, it is accomplished by re-synthesizing larger pages by physical arrangement of standard-sized pages. The atomic unit of mapping and permissions remains the Coyotos page size. Pages always appear in both the physical and virtual memory maps at naturally aligned addresses.

A device page may be any size 2^k that is greater than or equal to the basic page size. If they are larger than the basic page size, they appear in both the physical and virtual memory maps aligned at a 2^k address. Device pages may be marked non-cacheable or write-through.

A page capability may be inserted into the address space slot of a process, with the effect of defining an address space having valid offsets between [*guard*,*guard*+*pgsize*-1]. Attempts to reference offsets outside this range result in an invalid address exception.

Capability pages are byte-addressable units. However, capabilities must be stored and referenced at naturally aligned (16 byte) boundaries.

Address translation of an address *addr* with respect to a page or cappage capability is defined as follows:

1. If the value of *addr* exceeds the page size, an `InvalidAddress` exception is generated.
2. Otherwise: the *addr* is a valid offset, and the overall address reference is valid.

4.3 Address Space Composition

Address spaces are composed by means of the GPT object. A GPT is simply a fixed-length vector of capabilities (currently 16), each of which is paired with a **guard**. In Coyotos, the guard has been incorporated into the capability format itself. The state of a GPT is shown below.

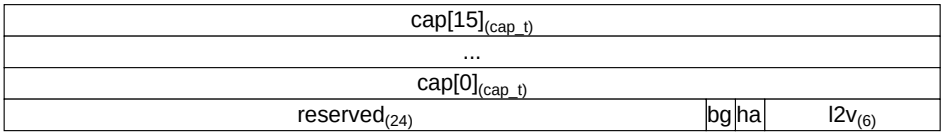

Figure 4.1: GPT State

Invariant: `l2v` $\geq$ `log2(`*`page size`*`)`

The meanings of the GPT fields are:

l2v Subspace size Each slot of the GPT names a subspace of size $2^{\texttt{l2v}}$ bytes.

ha fault handler Slot 15 of the GPT contains an Entry capability to the fault handler.

Care should be taken to set the `l2v` value appropriately when the `ha` bit is set. If the translation algorithm traverses an Entry capability in the normal course of translation, a malformed space exception will be generated.

bg background space Slot 14 of the GPT contains a memory capability to a background space (see window capabilities).

Care should be taken to set the `l2v` value appropriately when the `bg` bit is set. If the translation algorithm traverses a background capability during the normal course of translation, the translation result will appear as if a larger space was entered.

cap[0..15] Capabilities to subspaces.

When the `ha` or `bg` bits are set, it is the responsibility of the process managing the GPT to ensure that the `l2v` value prevents collision.

4.3.1 Translation Algorithm

Note: In the discussion that follows, it may be useful to refer to the capability representation for window and memory object capabilities (see Chapter 2), with particular reference to the `l2v` and `l2g` fields.

Address translation is performed by translating an unsigned virtual address `va` with respect to some memory capability `C` (a GPT, page, capability page, or window capability). Translation begins at the address space capability of the process structure with a 64-bit virtual address. In the normal case, the progress of translation causes bits to be "consumed" from the left, leading to virtual addresses of progressively smaller magnitudes. Window capabilities, however, may cause the remaining virtual address to grow as translation proceeds.

The virtual address `va` that is currently being translated is conceptually divided into three fields *g*, *u*, and *v*. The `g` field (which may be zero width) contains the **guard** value. The *u* field contains the index value that will be used to index into the next GPT. The `v` field contains either the address bits that will remain to be translated when the current step has completed (GPT or window capability) or the page offset bits (page or capability page capability).

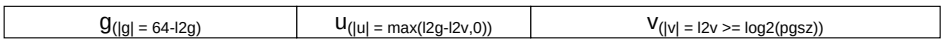

Figure 4.2: Virtual address structure

In reading the following section, recall the invariants described in Section 2.1.1. These are checked at capability fabrication time, and are assumed to hold by the following algorithm statements.

The values of *g*, *u*, and *v* are computed from the capability `C` and the address `va` as follows:

```
g = va >> C.l2g;
guard = C.guard << C.l2g;
```

```
u = (va - guard) >> C.l2v;
v = va & ((1u << C.l2v) - 1);
```

At the start of translation, the background space capability $C_{background}$ and the memory handler capability $C_{handler}$ are initialized to the Null capability, the virtual address is as provided by the hardware (or possibly the IPC logic) and the effective access restrictions AR is the empty set. Translation proceeds by iteration, with each iteration performing the following steps in sequence:

1. The g value is compared to the zero-extended guard value stored in the capability. If they do not match, the address is invalid and an InvalidAddress exception is generated.
2. If the u value exceeds the number of slots in the GPT, the address is invalid and an InvalidAddress exception is generated.
3. The effective access restrictions are updated from the capability C by:

   ```
   AR:=AR+C.restr
   ```

   If the resulting effective access restrictions are insufficient for the requested access type, an AccessViolation exception is generated.
4. Processing now proceeds according to the capability type:
   - If the capability type C.type is Page or CapPage, translation has completed successfully.
   - If a local or background window capability appears in the address space slot of a process, all addresses are deemed invalid.
   - If the capability is a local window capability appearing within some GPT, translation proceeds from the capability contained in the *rootSlot* slot of the GPT containing the local window capability at the offset named by the capability.

     ```
     va := v + C.offset
     C := containingGPT[C.rootSlot]
     ```

     Note that the invariants of Section 2.1.1 guarantee that there is no bitwise overlap between v and C.offset. That is: the addition can be correctly implemented as a bitwise "or" operation.
   - If the capability is a background window capability, translation proceeds from the capability to the background space with

     ```
     va := v + C.offset
     C := C_background
     ```

     Note that the invariants of Section 2.1.1 guarantee that there is no bitwise overlap between v and C.offset. That is: the addition can be correctly implemented as a bitwise "or" operation.

     Recall that $C_{background}$ is initialized to Null at the start of translation. If no other background capability has been defined at the point where the background window capability is encountered, all addresses that fall within the background window are invalid.
   - If the capability type is GPT, translation proceeds with

     ```
     gpt := target-of(C);
     if (gpt->bg)
       C_background = gpt->cap[14];
     if (gpt->ha)
       C_handler = gpt->cap[15];
     va := v
     C := gpt->cap[u]
     ```
   - If the capability is a Null capability, an InvalidAddress exception is generated.
   - Otherwise, a MalformedSpace exception is generated.

4.3.2 Exception Handling

If an exception is generated by the translation mechanism, and the memory exception handler capability `C_handler` is not `Null`, then the exception will be delivered to the memory exception handler. Otherwise, the exception type and address are stored in the process's `faultCode` and `faultInfo` slots, respectively, and the process is set running with the pending fault code, and the exception is then delivered as described in Section 3.3.1.

4.3.3 Cycle Detection

It is possible for an erroneous or hostile program to arrange GPT objects in such a way as to create a static cycle. Such an address space is *malformed*, and attempts to traverse such a cycle during address translation result in an `MalformedSpace` exception.

No final selection has been made for a method of cycle detection. Three rules have been proposed:

1. A bound on the total number of GPT structures that will be visited before generating a `MalformedSpace` exception.

   This method has been rejected. It has the unfortunate property that existing, valid addressing structures can be rendered invalid by "splitting" an existing GPT. We want to preserve the ability to split without semantic alteration in order to be able to map subspaces.

2. A bound on the total number of capabilities *that do not translate new bits* that will be visited before generating a `MalformedSpace` exception.

   This method keeps track of $|v_{least}|$, the shortest virtual address that has been obtained by translation to the current point. If $C.l2v \geq |v_{least}|$, then the current capability does not translate new bits.

   This method has been rejected. It has the unfortunate property that existing, valid addressing structures can be rendered invalid by "splitting" an existing GPT. We want to preserve the ability to split without semantic alteration in order to be able to map subspaces.

3. A bound on the total number of *bits* visited for translation, defined as the cumulative sum of $(|va|-|v|)$ for all capabilities visited during a translation attempt.

   This approach preserves the possibility of a correctness-preserving split operation.

All methods of cycle detection introduce a complication for implementers: the validity of addresses within a subspace is contextually dependent on the number of bound-countable events in the prefix path leading to that subtree. This means that two process address spaces may both have some subspace mapped at otherwise valid subspaces addresses, and selected subranges of the mapped subspace may nonetheless be valid in one space but not in the other.

Because of this problem, care must be taken when implementing page table sharing to ensure that page tables are shared only when all possible references through that hardware table are equally valid in all referencing contexts. If this is not done, one process would be able to produce valid mappings in the hardware mapping table that would be usable by the second, even though the second lacks the ability to produce those hardware mappings for itself.

4.4 Address Space Splitting

> **Experimental**
>
> The feature described in this section is experimental. It is not presently implemented, and may be removed in future versions of Coyotos.

In order to support the subspace transfer item described in the capability invocation chapter, Coyotos introduces a new type of exception that may occur in an address space: the `SplitFault`.

Split faults allow an invoker to send a single capability to an arbitrary 2^k page region of an address space, provided that the region is naturally aligned and the invoker has sufficient access rights to extract the dominating capability. Similarly, they permit a receiver to generate appropriate "holes" into which such a capability must be received.

The problem solved by split faults is that there may not be any naturally dominating GPT for the subspace. For example, in a system having 4 kilobyte (2^{12} byte) pages, the invoker may wish to transmit a 2^{11} page (2^{23} byte) subspace, but the subspace may currently be dominated by a GPT having `l2v=21`. That is: there is no single slot in the GPT that directly holds a capability of the desired span. Before a single dominating capability can be sent, this GPT must be "split" into an arrangement where the target subtree has a single dominating GPT with `l2v=23`. When such a send is attempted, the invoker will receive a `SplitFault` exception. This is an advisory that the GPT must be split in order to bring a dominating GPT into existence.

Similarly, if a receiver specifies a "hole" of some size $2^{\texttt{hlsz}}$ pages, there must exist some GPT in the receiver tree that could receive (with an appropriate guard value) a capability dominating a tree of the requested size.

The reason this feature is considered experimental is that the correct strategy for splitting GPTs is not obvious.

The address space splitting idea is not yet fully developed. There are certainly holes, including necessary but undefined exception types, that need to be resolved in the definition above.

Chapter 5

Capability Invocation (including IPC)

Coyotos is an object-based system. A process wishing to perform an operation (equivalently: invoke a service) does so by invoking some capability that it holds. The capability has a defined interface that specifies some set of invokable methods, including their argument and return types. The provider of these methods may be either the kernel or an application; the invocation mechanism is the same in either case. That is: Coyotos is an extensible object system [16]. The primary system call in Coyotos is the "invoke capability" system call (Section 6.5). Other system calls defined by the Coyotos specification may all be viewed as convenience wrappers for capability method invocations.

Because kernel-provided and application-provided services share a common invocation mechanism, it is necessary to specify both the low-level binding of capability interfaces and the externally observable semantics of capability invocation. While the specific binding is architecture-dependent, this chapter includes recommendations on bindings that suffice for most platforms.

The invoke capability system call implements a variant of the *SendAndWait* primitive proposed by Liedtke [12] or the CALL and RETURN primitives of EROS [4]. The send phase of the invocation can be blocking or non-blocking. If a non-blocking send is performed, some or all of the message may be truncated. The receive phase may wait for an arbitrary endpoint (an "open wait") or a specific endpoint (a "closed wait"). The receive phase is optional.

5.1 Invocation Payload

An invocation passes a message that consists of:

- Up to 8 direct words, the first of which is the invocation control word. The size of these words is architecture dependent. These words may be carried in registers or memory, as specified by the architecture-specific annex for the target platform. The index of the last word transmitted is given by `IPR0.ldw`.

  Input parameter word 0 of the invoke capability operation contains control information describing the rest of the message payload:

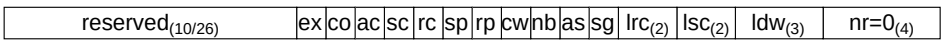

Figure 5.1: Invocation control word (input)

  Provided the invokee is valid and well-formed, a message consisting solely of untyped parameter words is guaranteed to proceed without exceptions on all architectures.

- Up to four capabilities. Capabilities are transmitted if `IPR0.SC=1`. If so, `IPR0.lsc` gives the index of the last capability transmitted. Capabilities are received if `IPR0.AC=1`. If so, `IPR0.lrc` gives the index of the last capability that will be accepted.

- An **indirect string** of up to 64 kilobytes. The length of this string is given in a parameter word to the system call.

In addition to the payload of the invocation, the invoker specifies:

- The capability to be invoked.
- Whether they are willing to block in order for the message to be delivered (`IPR0.NB`).
- Whether to fabricate a reply capability (`IPR0.RC`).
- Whether the receive phase should be performed (`IPR0.RP`).
- Whether copy-out of soft registers should be performed on those architectures that define soft registers. (`IPR0.CO`).
- Whether the receive phase should accept messages only from a particular endpoint (`IPR0.CW`, `upcb.rcvEpID`).

If a receive phase is executed, the receiver receives the following information in addition to the invocation payload:

- The endpoint identifier of the Endpoint on which the invocation was received.
- The "protected payload" of the capability that was invoked.
- The length of the string that was sent, it any.
- A modified copy of the invocation control word, which indicates various information about the incoming message. In this returned word, the `u`, `RC`, and `SC`, fields are copied from the sender's *input* invocation control word. The `lsc` field indicates the number of capabilities that have been received. The `ldw` field indicates the number of data words that have been received.

The protected payload and the endpoint ID can be used to determine the receiver-defined context in which the received message should be interpreted. One common use of these fields is for the endpoint ID to identify the object invoked and the protected payload to identify the permissions on that object.

5.2 Invocation-Related Exceptions

Exceptions may occur during invocation on either the sender or the receiver side of the transmission. All such exceptions logically occur *before* the invocation. In practice, exceptions are generated as a consequence of payload transfer. If an exception occurs, the implementation is free to resume the transfer at the point of interruption if it is able to do so. However, the receiver of an interrupted transmission logically reverts to the beginning of its receive phase when an exception occurs. In the event that a second sender is attempting to send when a messaging exception is incurred, the second sender's message may prevail.

If the sender specifies non-blocking transmission, the transfer of indirect strings and capabilities is "best effort." If the receiver incurs a page fault during the receipt of an indirect string or a capability argument, that argument will be truncated. In this case the receiver will be notified of truncation, but no receiver-side exception will be generated.

The meaning of a non-blocking send is that the sender is unwilling to be blocked for any cause whose handling is controlled by the receiver. The use-case for this option is a server returning a reply to an untrusted client. For purposes of understanding truncation, a hardware page fault that is successfully resolved by the object paging subsystem is not considered to be an architecturally observable fault. Similarly, an exception that can be satisfied by reconstructing a hardware mapping entry from an already defined GPT hierarchy is not considered an architecturally observable fault.

5.3 Endpoints

A process that wishes to accept capability invocations does so by means of one or more endpoints (Figure 5.2). Endpoints have two capability types: the Endpoint capability, which implements the control interface for the endpoint object, and the Entry capability, which provides the means for extending the object system. When an Entry capability is invoked, the invocation parameters are delivered as a message to the process named by the `recipient` field of the Endpoint.

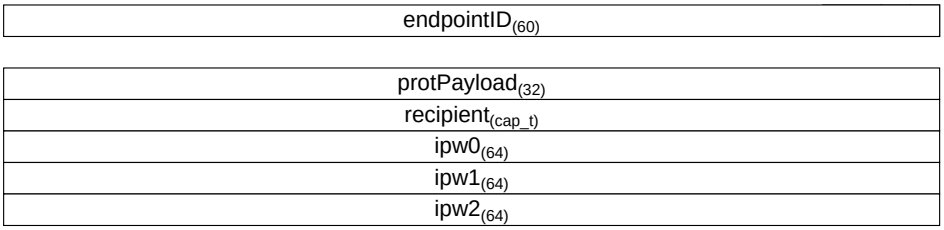

Figure 5.2: Endpoint structure

The meanings of the endpoint fields are:

Field	Meaning
pm	**Payload Match** Indicates that the protected payload of the endpoint should be compared to the protected payload of the Entry capability. If they are not equal, the invocation behaves as if the Null capability had been invoked.
protPayload	**Protected Payload** This value will be conditionally used as a matching value if `PM` is set (1).
endpointID	A 60-bit field having meaning only to the recipient. The value of this field will be delivered to the recipient during message receive.
recipient	A process capability to the receiving process.

Non-Normative Illustration

When a server implements a single logical object, it will typically operate with two endpoints. The first is the one used to invoke the service (the "receive endpoint"). The endpoint ID of this endpoint is not used. The protected payload of the corresponding Entry capability may be used to express distinct permissions or restrictions on the permitted operations. The second endpoint is used to accept replies (the "reply endpoint"). The endpoint ID of this endpoint is used as a matching value to implement a closed wait so that unrelated messages are not received where a reply is expected. The endpoint's protected payload is used to ensure that no more than one reply will be received (by means of the `IPR0.RC` bit of the invoke capability system call).

When a server implements multiple objects, a distinct receive endpoint is typically allocated for each object implemented by the server. In this case, the endpoint ID is used to identify which object or service is being invoked, and the protected payload field of the corresponding Entry capability is used to express distinct permissions or restrictions on the permitted operations on that object.

Non-Normative Note on Reply Endpoints

If the `IPR0.RC` bit is set in the invocation control word parameter, the protected payload of the endpoint is pre-incremented before the Entry capability is fabricated. The purpose of the RC bit is to allow a caller to ensure that a call/return sequence receives at most one reply in the normal case. This is accomplished by ensuring that stale reply capabilities are invalidated (by protected payload mismatch) before the next receive on the reply endpoint is performed.

Whenever an Entry capability is invoked, the invokee receives the protected payload value of the invoked Entry capability. In the case of a reply endpoint, the PM bit is set, so the received protected payload value matches the value stored in the endpoint.

It is the responsibility of the application to notice when the incoming protected payload value approaches UINT32_MAX. In this situation, the pre-increment will overflow the protected payload counter when it is next used. The recommended solution for this is to obtain a new reply endpoint from a space bank when the protected payload reaches UINT32_MAX-1.

5.4 Semantics of Kernel Capability Invocation

To ensure consistent invocation behavior, it is necessary to specify the externally observable behavior when a kernel-implemented method is invoked. In particular, the observable effect on process state and the sequencing of operations and events during a kernel invocation must be defined.

When a kernel capability is invoked, the externally observable behavior should be as if the invoker had invoked an endpoint to some application providing the service. Because no kernel operation accepts an indirect string, the invocation of a kernel capability behaves as if this hypothetical provider had performed a receive phase with `IPR0.AS=0` (no strings will be accepted). This hypothetical provider arrives at the specified answer and accomplishes any effects of the invocation by unspecified means. It then replies as if it had invoked the `InvCap` system call with the control bits of the first input parameter word set as follows: `NB=1` (non-blocking), `RC=0` (no reply capability is generated), `CW=0` (the kernel conceptually enters an open wait state), and `RP=1` (the kernel waits for the next invocation). In addition, the `SC` (send capabilities) control bit will be set (clear) if capabilities are (are not) returned by the method. Note that because the kernel reply is non-blocking, and the kernel is deemed to be in the *running* state until it has replied, the reply from a kernel-implemented capability cannot cause a second kernel-implemented capability to be invoked.

This statement of behavior has (at least) the following implications:

- The effects of a kernel capability invocation occur whether or not the invocation returns successfully, provided any preconditions specified for the method are satisfied.
- There exist several kernel operations that alter the state of a process. When the process altered is also the process receiving the kernel reply (the "invokee"), the kernel behavior must be well-defined. There are two such cases:
    1. The invokee process is destroyed as an effect of the invoked method. In this case, the reply proceeds as if via an endpoint that contains a Null capability.

By intention, kernel-implemented operations satisfy two invariants that simplify or eliminate other potentially obscure corner cases:

- No kernel-implemented interface accepts or returns an indirect string.
- Kernel methods that modify address space mappings or revoke objects return only scalar return values (and therefore behave as if `SC=0`). This ensures that changes in the meaning of the receive capability `capitem_t` values cannot impact the return of these operations.

Undefined Locations The content of receive buffers, receive parameters, and receive capability locations is undefined between the start of the IPC receive phase and the completion of the IPC receive phase. For performance reasons, the kernel is entitled to arbitrarily modify state whose content is undefined during invocation. In particular, the kernel is entitled to modify the receive parameters or the receive string buffers of a waiting process without releasing that process from its wait state. This allows the kernel to more efficiently implement indirect string moves that may induce invoker or invokee page faults during the transfer. This means that *all* of the receive string buffers of a recipient may be modified during receive, even if the final message received sends only a single indirect byte. Similarly, any valid receive capability locations may be overwritten even if a smaller number of capabilities was transferred.

State Transitions The overwhelming majority of kernel capability invocations return to the invoker without generating any exception. In these cases, the kernel may behave as if the operation occurred instantaneously, with the consequence that the invoker may never be observed to leave the *running* state.

Elided Reply Capability When a kernel capability is invoked and replies to the invoker without an exception, the kernel implementation is free to elide the fabrication of the reply capability. Elided reply capabilities are observable because the protected payload value of the reply endpoint will not be incremented.

Chapter 6

System Calls

Coyotos currently defines three system calls:

Number	Name	Description
0	`InvokeCap`	Invokes a capability and (optionally) waits for a reply on an endpoint.
1	*reserved*	Reserved for future use.
2	`CopyCap`	Copy a capability from one location to another.
3	`Yield`	Yield the processor, moving the current process to the back of its scheduling class.
4..15	*reserved*	Reserved for future use.

6.1 Parameters and Parameter Words

At the system call trap interface, arguments and return values are conveyed by means of a combination of registerized parameter values and (optionally) a system-call specific stack frame. Every architecture-specific annex specifies a subset of hardware registers that are be used to convey system call parameters. No annex defines fewer than four registers to be available at this interface. Where a specialized stack frame is specified, the architecture-specific annex may specify that some or all of that stack frame is conveyed across the user/supervisor boundary in registers. The corresponding fields of the stack frame will never be accessed by the kernel.

The Coyotos system call specification ensures that all arguments and return values of system calls *other than* `InvokeCap` can be marshalled in registers. In addition, the majority of kernel-implemented capabilities, including all capabilities likely to be invoked in performance-critical application paths, can be invoked without a string parameter.

In the system call specifications that follow, the notation IPR*n* and OPR*n* indicate input and output parameter registers, respectively. Except where required by the architecture-specific system call mechanism, or explicitly noted by the system call, output registers retain their value at the time of system call entry.

6.2 Exceptions

The following exceptions may be incurred by the caller during system call execution.

Exception	Cause
`MalformedSyscall`	The operand was malformed. This includes field value range errors or reserved type codes. The `faultInfo` field is zero.
`MisalignedReference`	The operand specified a capability address, but the address described is not aligned to a 16-byte boundary. The `faultInfo` field contains the errant address value.
`InvalidAddress`	The operand specified an address that is not defined. The `faultInfo` field contains the errant address value.
`AccessViolation`	A store operation was attempted, but the operand specified an address that does not permit write access. The `faultInfo` field contains the errant address value.
`DataAccessTypeError`	The address specified by a data load/store operand does not reference a data page. The `faultInfo` field contains the errant address value.
`CapAccessTypeError`	The address specified by a capability load/store system call does not reference a capability page. The `faultInfo` field contains the errant address value.
`MalformedSpace`	The address specified by the operand violated the well-formed address space constraints. The `faultInfo` field contains the errant address value.

Any system call may generate the `MalformedSyscall` exception if bits marked "reserved" are non-zero or specified field value bounds are exceeded. Individual system call descriptions below specify which of the other exceptions may be incurred by that system call.

6.3 Capability Locations

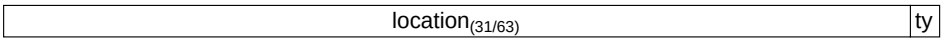

Figure 6.1: caploc_t structure

A `caploc_t` parameter (Figure ***?MISSING XREF ID?***) describes a generalized capability location that is either a capability register (ty=0) or a memory address (ty=1).

The encoding of register `caploc_t` values is identical to the encoding used for `capreg_t` values, modulo the wider `location` field. When the `caploc_t` describes a memory address, the `location` field holds the most significant bits of the address. Capability addresses are required to be 16 byte aligned. In consequence, the expression of valid capability addresses is not restricted by the re-use of the least significant bit for this purpose.

The size of a `caploc_t` matches the architecture-defined word size.

6.4 Pseudo-Instructions

The `Yield` and `CopyCap` system calls are best thought of as pseudo-instructions.

6.4.1 Yield [syscall]

The `Yield` system call relinquishes the processor. If the `I` bit is clear (0), the yielding process is placed at the end of the appropriate ready queue.

Parameter	Format	
IPR0	reserved$_{(28/60)}$	NR=4$_{(4)}$

The `Yield` system call does not return any output parameters.

Open Issue

Should there be a directed yield operation? If so, the yield system call needs to take a second parameter.

6.4.2 CopyCap [syscall]

Parameter	Format	
IPR0	reserved$_{(28/60)}$	NR=2$_{(4)}$
IPR1	source$_{(caploc_t)}$	
IPR2	dest$_{(caploc_t)}$	

The `CopyCap` system call copies a capability from a source location (register or memory address) to a target location (register or memory address). The `CopyCap` system call does not return any output parameters. Exceptions may be generated by the references to the *source* and *dest* parameters.

Any of the exceptions listed in Section 6.2 other than `DataAccessTypeError` may be generated by this system call.

6.5 InvokeCap [syscall]

The `InvokeCap` system call invokes a capability, passing the supplied parameters to the implementing server. It is both the most complex and the most commonly used system call in the interface. It invokes one capability and optionally blocks for an incoming message on an Endpoint. `InvokeCap` takes a variable number of parameters determined by the invocation control word provided in IPR0 and a system-call specific stack frame.

Any of the exceptions listed in Section 6.2 may be generated by this system call.

6.5.1 Arguments

The arguments to `InvokeCap` are organized in a specialized stack frame. Some elements of the stack frame are then registerized. The canonical stack frame format for `InvokeCap` is shown below:

```
typedef struct {
  uintptr_t pw[8];
  union {
    caploc_t  invCap; /* on entry */
    uintptr_t pp; /* on exit */
  } u;
  caploc_t  sndCap[4];
  caploc_t  rcvCap[4];
  uint32_t  sndLen; /* FIX: Should be 16 bits */
  uint32_t  rcvBound; /* FIX: Should be 16 bits */
  void     *sndPtr;
  void     *rcvPtr;
  uint64_t  epID;
} InvParameterBlock_t;
```

All architectures guarantee that the `pw[0]..pw[3]` field are conveyed in registers on both entry and exit. Since the message payload may contain up to 8 data words, the caller should assume that `pw[0]..pw[7]` may be modified on return.

The control word parameter to `InvokeCap` is given by:

Argument	Format
IPR0	reserved$_{(10/26)}$ | ex | co | ac | sc | rc | sp | rp | cw | nb | as | sg | lrc$_{(2)}$ | lsc$_{(2)}$ | ldw$_{(3)}$ | nr=0$_{(4)}$
IPR1	Input `pw[1]`, if transmitted.
IPR2	Input `pw[2]`, if transmitted.
IPR3	Input `pw[3]`, if transmitted.
OPR0	reserved$_{(10/26)}$ | ex | co | ac | sc | rc | sp | rp | cw | nb | as | sg | lrc$_{(2)}$ | lsc$_{(2)}$ | ldw$_{(3)}$ | nr=0$_{(4)}$
OPR1	Output `pw[1]`, if received.
OPR2	Output `pw[2]`, if received.
OPR3	Output `pw[3]`, if received.

Other fields and registers are sourced and modified according to the interpretation of the control word and the parameters provided in the stack frame.

The fields of the control word have the following meanings:

Bit	**Name**	**Meaning**
ldw	Last Data Word	On entry, the index of the last data word transmitted. Following a receive phase, this field will contain the values specified in the sender's invocation control word. A receiving process must be prepared to accept 8 direct data words.
lsc	Last Sent Capability	The index of the last sent capability. No capabilities are sent if the `SC` field is clear (0). Following a receive phase, this field will contain the value specified in the sender's invocation control word.
lrc	Last Received Capability	On entry: the index of the last capability slot that will be received. No capabilities are received if the `AC` field is clear (0).
sg	Send Gather	Send transmitted string(s) using the gather mechanism. This mechanism is not yet specified, and the bit should be clear (0) in all invocations.
as	accept scatter	Accept transmitted string(s) using the scatter mechanism. This mechanism is not yet specified, and the bit should be clear (0) in all invocations.
nb	Non-Blocking	If set (1), send is non-blocking. Any action that would require the sender to be enqueued in such a fashion that re-awakening is controlled by the receiver will result in a dropped message. Any action that would cause a receiver-controlled exception handler to be executed will result in a truncated message.
cw	Closed Wait	If set (1), indicates that the receive phase is performing a closed wait, and only messages from endpoints whose endpoint ID matches the supplied `epID` value will be accepted. If clear (0), no restrictions are imposed on receipt and the `epID` field is ignored.
sp	Send Phase	If set (1), the send phase will be executed.
rp	Receive Phase	If set (1), the receive phase will be executed.
rc	Reply Capability	If set (1), and `sndCap0` names an endpoint capability, `sndCap0` will be replaced with the corresponding Entry capability. The protected payload of the Entry capability will be set to the current protected payload value stored in the endpoint.
sc	Send Capabilities	If set (1), capabilities will be transmitted. Following a receive phase, this field will contain the value specified in the sender's invocation control word.
ac	Accept Capabilities	If set (0), incoming capabilities will be accepted.
co	Copy Out	If set (0), the copy out phase will be executed. Soft register values will be copied back from the kernel to the invocation parameter structure.

Bit	Name	Meaning
ex	exceptional return	If set (1), indicates that the message payload describes an exception. Following a receive phase, this field will contain the value specified in the sender's invocation control word.

The parameters of the `InvokeCap` system call are unusual, in that IPR0 determines how the remaining input parameter words are interpreted. Further, some of the bits in the IPR0 word determine how the output parameteres are processed and delivered. `InvokeCap` is *also* unusual because control flow may not return immediately to the invoker. Finally, it is unusual because an `InvokeCap` system call may cause exceptions to be incurred by the receiver *during* the system call rather than before it.

Conventions

`InvokeCap` is a reflective system call. Because the kernel-implemented capabilities use the same request marshalling and demarshalling conventions as application-implemented capabilities, these conventions are effectively mandated.

Requests `IPR1` contains the request code (a method code that is typically assigned by CapIDL). The implementing process uses this request code, in combination with the received endpoint ID and protected payload, to determine what interface has been invoked, what method has been requested, and what permissions the invoker has. By convention, no method should be assigned the method code zero (0).

Replies If `OPR0.ex` is clear (0), then the remaining parameter words contain the specified out parameters according to the interface specification. If `OPR0.ex` is set (0), then the remaining parameter words contain the exception message, which consists of a 64-bit exception code followed by an optional structure containing additional information. The exception code may occupy `OPR1` or `OPR1..OPR2` or `OPR2..OPR3` (notably SPARC) depending on whether the target architecture is 64-bit or 32-bit and the alignment restrictions imposed on 64-bit integral values by the underlying architecture.

Kernel Invocation Conventions

The following behavioral specification describes the semantics of the `InvokeCap` instruction as if the capability invoked were an Entry capability. If the capability invoked is a kernel-implemented capability, the invocation behaves as if the kernel were a receiving process that had just performed a SendAndWait with IPR0 and IPR1 fields set as follows:

Field	Meaning	Field	Meaning
IPR0.CW=0	Perform an open wait.	IPR0.NB=0	Reply will be non-blocking.
IPR0.AS=1	Accept string arguments.	IPR0.AC=1	Accept capability arguments.
IPW1	(ignored `capitem_t`).	IPR0.RP=1	Perform a receive operation.
IPR0.RC=0	No reply capability.	IPR0.ldw	Set according to last operation.

6.5.2 Return Values

The return values of `InvokeCap` are copies of the sender's argument values with minor alterations:

- The protected payload of the invoked endpoint capability is supplied in place of the invoked capability `caploc_t`.
- The sender control bits are replaced by `T`, a bit indicating that one or more incoming typed items were truncated.
- All addresses supplied in the input typed items are zeroed in the output typed items.
- On output, the kernel injects an `epiditem_t` as the first output typed item.

Result	Format
OPR0	reserved$_{(10/26)}$ | ex | co | ac | sc | rc | sp | rp | cw | nb | as | sg | lrc$_{(2)}$ | lsc$_{(2)}$ | ldw$_{(3)}$ | nr=0$_{(4)}$
OPR1	rcvPP$_{(32/64)}$
OPR2..OPR*u*	untyped word$_{(32/64)}$
OPR*u*+*1*..OPR*u*+*t*+*1*	typed item words$_{(32/64)}$

Chapter 7

Schedules

7.1 Scheduling Model

Chapter 8

Other Kernel Objects

This chapter describes the services provided by the miscellaneous kernel capabilities.

The `null` capability is used when a non-optional capability field must be transmitted but the sender does not wish to send a capability in that position. Capabilities to destroyed objects become null.

The `keybits` capability discloses the canonical representation of capabilities. The `keybits` capability is considered sensitive, and should be closely held. The value of the hazard bit **HZ** is always shown as zero.

The `discrim` capability classifies capabilities into one of a limited set of classifications. The purpose of `discrim` is to support the implementation of the confinement policy by the `constructor`, which is one of the core Coyotos applications.

The `range` capability conveys the authority to fabricate and destroy arbitrary object capabilities. The `range` capability is highly sensitive, and should be closely held.

The `sleep` capability allows its holder to receive an event at a scheduled time.

The `IrqCtl` capability allows the holder to allocate `IrqWait` capabilities. This capability is highly sensitive, and should be closely held.

The `IrqWait` capability allows the holder to wait for a particular interrupt to occur. This capability is highly sensitive, and should be closely held.

The `schedctl` capability allows the holder to alter the kernel-level scheduling dispatch table. This capability is highly sensitive, and should be closely held.

The `checkpoint` capability allows the holder to initiate a system-level snapshot operation and force the checkpoint age-out logic to run to completion. This capability is highly sensitive, and should be closely held.

The `obstore` capability implements a “reverse” protocol. The object store server uses this capability to wait for kernel object fill and flush requests and acts on them.

Part II

Microkernel Realization

Part I of this specification describes an abstract machine, and gives semantics for the objects associated with that machine. While there are exposed implementation dependencies that are inherent in the representations of process register sets and the sizeing of page objects, the theory of operation for these objects is relatively independent of the underlying hardware.

As a practical matter, real machines incorporate features beyond these core abstractions. These are not managed by the microkernel, but they require some degree of support in the specification. In particular, the specification must address issues of device memory and support for direct memory access (DMA), because these are places where the binding between abstract and concrete objects becomes visible to software.

In addition, the Coyotos specification must address the mechanism of persistence for those implementations that provide it.

Chapter 9

Device Interaction and DMA Support

This section describes behavior that is not yet implemented.

This chapter addresses support for device access and for localizing pages so that DMA hardware features can be used. This chapter does *not* address kernel support for integrating hot-pluggable memory cards as general-use object cache storage.

9.1 Device I/O Registers

Some hardware implementations provide a hardware register address space (I/O space) that is independent of the memory address space. This is most notably true on the Pentium, but more selectively true on other processors (e.g. Coldfire MOVCR). Authorization of access to the hardware register address space is determined by the per-process I/O space capability. The mechanics of hardware control register access is inherently target-dependent, and the mechanism used should be specified by the architecture-specific annex.

On some architectures, device registers can be memory mapped, and page-level protections are sufficient to provide effective isolation. On those systems, device registers can be treated as a form of device memory.

9.2 Device Memory

Some devices publish memory to the device bus that can be mapped wholly or partially into the main memory address space. Modern video cards are an example of this.

Typically, the mechanical and electrical bus architectures of a given machine limit the number of devices that are able to publish memory-mapped device memory. Busses such as SCSI, ATAPI, USB, and FireWire, by contrast, generally do not support memory mapped interfaces for their attached client devices. The controllers themselves may use memory mapping, but the attached devices do not.

This design allows the kernel to pre-plan for memory attached device memory by reserving a pre-planned constant number of page frames for each device, where the total number of memory-attachable devices is bounded at compile time. The *size* of these pages may be larger than the hardware page size defined by the MMU.

9.2.1 Device Ranges

A **device range** describes a contiguous region of the physical memory map that is backed by some memory-attached device. Every device range may have up to 16 data pages, each of which is of some size 2^k. Thus, a device range is defined as a (base, limit, size) tuple, where the `base` and `limit` are physical addresses and the `size` describes both the alignment restrictions on the `base` and `bound` addresses and the size of the physical pages defined by this

device range. A one megabyte memory-mapped device region can therefore be specified as 16 64Kbyte pages or as 4 256Kbyte pages.

The architecture-specific annex specifies the number of device ranges that can be defined in a given implementation. This is required because the kernel must be able to pre-reserve storage for device memory ranges during early initialization, long before the drivers associated with those ranges are executed or the size of the respective device memory spaces are known.[1]

Device memory ranges are not persistent.

The theory of operation for device ranges is that the kernel pre-reserves the storage for the device range descriptors at startup time, and also the per-page management data structures that are required to track the device pages that will later be defined. As individual drivers come up and determine their associated physical address windows for device memory, the "allocate" a device range for that device. Once the device range is defined, the coyotos.Range capability can be used to allocate device pages within that device range.

Destroying a device range destroys all of the device pages currently associated with that range.

9.2.2 Device Pages

A **device page** is a data page whose physical address falls within the range defined by some device range. The size of a device page may be any size 2^k that is greater than or equal to the hardware page size. The physical and virtual addresses of a device page are naturally aligned.

Device pages are not persistent, and are not subject to ageing.

> **Issue**
>
> I suspect that device page *capabilities* should not be persistent, and should be written to disk as Null.

The OID range [0xff00000000000000ull, 0xffffffffffffffffull] is reserved for device pages. If `l2page` is defined such that the hardware page size is 2^{l2page}, then the OID of the physical page whose physical address $2^{k \geq l2page}$ is (0xff00000000000000ull$+2^{k-l2page}$). The corresponding page capability is obtained by invoking Range.getCap or Range.waitCap with this OID and a type of `otPage`. The returned capability will be a page capability to page of size 2^k at the desired physical address.

While a device page capability obeys the page capability protocol, implements the same permissions as a page capability, and can be mapped like any other page capability, the term "device page" is somewhat misleading:

- Device pages are not persistent.
- A device page may map onto device registers rather than device memory.
- A device page does not really define a block of storage. It is a "view" onto memory that resides on some device. This view has a known location and size in the physical address space, but in some cases the device driver may be able to alter the relationship between this view and the associated device memory (e.g. by changing how addresses presented to the card are demultiplexed).
- Device pages are not zeroed on allocation or rescind, because we have no way to know what sits behind them. It may be registers, and the kernel does not know what the implications of zeroing those registers might be.

Because of this, writes or reads to device pages can have unusual side effects, and it is not always the case that a write to a location within a device page followed by a read at that location will produce the same value. In contrast to conventional pages, the semantics of device pages is wholly or partially determined by the device.

[1] In implementations where boot-time parameters can be provided to the kernel, an implementation is free to provide means to extend this default, but this should be viewed as a means to deal with high-end, customized hardware configurations rather than a means for adapting to card insertion and removal.

Several of these factors suggest that use of device pages should be undertaken with care, and should be restricted to device drivers. If direct access to device memory must be provided to persistent programs, the provided memory space should be encapsulated by an opaque GPT.

Device Page Windows It is possible to define windows into device pages using window capabilities, and the window size may be smaller than the device page size, but must be at least as large as the hardware page size. This permits a device driver to use larger device page sizes without sacrificing page protection at the normal hardware granularity.

9.3 Object DMA

Because device pages exist at known physical addresses, it is possible for device drivers to use physical direct memory access (DMA) mechanisms to move data to and from these pages. This does not address the problem of DMA to and from conventional pages. For example, it is desirable for drivers to be able to transfer data directly to and from user space.

In abstract, there are three issues that need to be addressed in order to support this:

1. Discovering the physical address at which DMA should occur to reach the target page.
2. In some cases, ensuring that the target page falls within the physical address span reachable by the specific DMA subsystem, and satisfies any alignment or address congruency requirements of that DMA engine.

   An ancillary problem here is that two DMA engines may compete for the same page with incompatible addressing constraints. This either needs to be prevented or resolved consistently.
3. Ensuring that the target of DMA remains in memory (pinned) until the DMA operation has completed.

Part III

Microkernel Interfaces

coyotos.AddressSpace

Abstract Interface `coyotos.AddressSpace`

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.AddressSpace
```

Synopsis: Operations common to all Coyotos memory-related capabilities.

The memory interface captures constant values and operations that are common to all memory-related capabilities.

Type Definitions

slot_t Slot index values.

```
typedef uint32 slot_t;
```

Exceptions

OpaqueSpace Unable to return a capability which is under an `OP GPT`.

NoSuchSlot No slot of the requested size was found.

Note that this exception can be raised by the `extendedFetch` and `extendedStore` operations if the requested slot location is a leaf slot and the leaf object is not a capability page.

CapAccessTypeError Unable to read capabilities from a Page.

Operations

getSlot Fetch capability from slot `slot`.

```
Cap getSlot(
    slot_t slot);
```

If the invoked capability is a weak capability, the returned capability will be the weakened form of the capability that was found in the target slot.

Raises `CapAccessTypeError` if the invoked capability designates a memory object that does not have capability slots.

setSlot Store cap `c` into the specified slot.

```
void setSlot(
    slot_t slot,
    Cap c);
```

Raises: NoAccess

Inserts a new capability into the GPT or page at slot. If the specified prefix already exists, the associated capability is overwritten.

Raises CapAccessTypeError if the invoked capability designates a memory object that does not have capability slots.

Raises NoAccess if the invoked capability is weak or read-only.

guardedSetSlot Store capability c into the specified slot, adjusting the guard according to guard.

```
void guardedSetSlot(
    slot_t slot,
    Cap c,
    guard_t guard);
```

Raises: RequestError

Raises NoSuchSlot if the invoked capability designates a memory object that does not have capability slots.

Raises RequestError if the argument capability c is not a memory capability.

fetch Fetch capability from the specified offset in address space.

```
Cap fetch(
    coyaddr_t offset);
```

Raises: NoSuchSlotNoAccess

This is the capability invocation form of the load capability instruction. The primary difference is that it can be invoked on any memory capability.

If the invoked capability is a weak capability, the returned capability will be the weakened form of the capability that was found in the target slot.

store Store c at the specified offset in address space.

```
void store(
    coyaddr_t offset,
    Cap c);
```

Raises: NoSuchSlotNoAccess

This is the capability invocation form of the store capability instruction. The primary difference is that it can be invoked on any memory capability.

Raises NoAccess if the invoked capability is weak or read-only.

extendedFetch Return capability from l2arg sized slot at specified offset.

```
Cap extendedFetch(
    coyaddr_t offset,
    l2value_t l2arg,
    OUT l2value_t l2slot,
    OUT restrictions perms);
```

Raises: OpaqueSpaceRequestError

Retrieves a capability from an address space from the last GPT whose l2v is greater than or equal to l2arg, subject to the following constraints:

- no GPT having an l2v less than l2arg will be traversed,
- no capability having an l2g less than l2arg will be traversed,
- any slot traversal that would result in a malformed or invalid space exception will not be performed – that is, a slot from the containing GPT will be returned.

On successful return, l2slot contains the l2v of the GPT containing the returned capability, and perms describes the cumulative permissions along the path up to and including the containing GPT, but *excluding* the permissions of the returned capability. If the access path traverses a weak capability, the returned capability will be the weakened form of the capability that was found in the target slot.

Note that if extendedFetch returns an l2slot that does not match the requested l2arg, and there are GPTs in the tree whose l2v is greater than their parent's l2v (larger space in small hole), a call to extendedStore(offset, l2slot, newCap) using the returned l2slot may replace a different slot then the slot extendedFetch returned.

If either the invoked capability or the GPT that it names do not satisfy the traversal constraints described above, the requested slot does not exist. The invoked capability is returned, an l2slot of zero is reported, and an perms of zero is reported.

Raises OpaqueSpace if the traversal encounters an opaque capability on the path up to, but excluding the returned capability.

Note: This method specification is provisional. We are attempting to expose existing kernel function for use by memory fault handlers, but we probably haven't got the return values and termination conditions quite right yet.

extendedStore Store value into a l2arg sized slot at offset offset.

```
void extendedStore(
    coyaddr_t offset,
    l2value_t l2arg,
    guard_t guard,
    Cap value);
```

Raises: OpaqueSpaceRequestErrorNoSuchSlotNoAccessCapAccessTypeError

Traverses the address space headed by the invoked GPT to find a slot whose size is l2arg. The request will fail if:

1. The offset presented is not a multiple of 2^l2arg, in which case RequestError is raised.
2. The l2g field of a traversed capability is less than l2arg; NoSuchSlot may be raised.
3. A guard mismatch occurs; NoSuchSlot may be raised.
4. An opaque (OP) GPT capability is traversed; OpaqueSpace may be raised.
5. The slot number we are traversing to in a GPT is out of range. NoSuchSlot may be raised.
6. The l2v of a GPT we traverse is less than l2arg; NoSuchSlot may be raised.
7. A ro or wk capability is traversed; NoAccess may be raised.
8. An l2arg less than COYOTOS_PAGEL2V is given. RequestError will be raised.

If the traversal succeeds, the capability slot modified will be the first slot encountered by the traversal algorithm such that the l2v value associated with the slot matches the argument l2arg. This location will be overwritten by value.

If the traversal fails, and more than one failure could be returned, it is not yet defined which failure is raised.

If the l2g component of guard is is non-zero, and the capability named by value is a memory capability, the supplied guard word replaces the one found in the capability named by value.

erase Rewrite this object to null capabilities and/or zero data.

```
void erase();
```

Raises: NoAccess

Raises NoAccess if the invoked capability is not writable.

copyFrom Copy this object from another object of the same type, returning a capability with a matching guard.

```
AddressSpace copyFrom(
    Memory other);
```

Raises: `NoAccessRequestError`

Raises `NoAccess` if the invoked capability is not writable. Raises `RequestError` if the passed capability type does not match the invoked capability type.

The returned capability will be a duplicate of the invoked capability, with the guard field copied from the target object. Restrictions are **not** copied from the target object. If the caller wishes to copy the elements of the target object without altering the guard field of the recipient node, the returned capability can be discarded.

coyotos.AppNotice

Interface coyotos.AppNotice

Derivation:

coyotos.Cap
  coyotos.AppNotice

Synopsis: Application-defined interrupt posting interface

An AppNotice capability authorizes up to 32 distinct notices that are guaranteed to be delivered to the recipient. Notices posted using the postNotice method are guaranteed to be delivered when the recipient (if any) next enters a receiving state, but invocation of postNotice is also guaranteed not to block. This makes the AppNotice capability useful to support the implementation of flow control in bidirectional streams or input notification in situations where the notifier must not block.

If a given notice is posted multiple times prior to receipt, the receiving application will receive it exactly once.

Operations

postNotice Post a (set of) application-defined notices notices n to the application.

```
/* oneway */ void postNotice(
    uint32 notices);
```

The postNotice method posts the authorized subset of notices to the receiving application if one exists.

There are two ways in which the invocation of postNotice can fail:

1. The endpoint directly named by the AppNotice capability may contain a Null capability in its recipient slot.
2. The endpoint itself may have been destroyed.

In both cases an error is returned internally, but cannot be received because a **oneway** invocation does not involve a receive phase.

getNotices Return the list of notices whose delivery is authorized by this capability.

```
uint32 getNotices();
```

coyotos.Cap

Abstract Interface `coyotos.Cap`

Synopsis: Operations common to all Coyotos capabilities.

The cap interface defines a set of operations that are common to all Coyotos capabilities. While some objects do not implement some of these operations (e.g. many kernel capabilities do not honor the destroy operation), these operations are nonetheless so universal that they warrant inclusion in the common ancestor of all interfaces.

Type Definitions

coyaddr_t Type of address space positions as defined by the operating system.

Coyotos defines address space offsets to be 64 bits on all platforms.

```
typedef uint64 coyaddr_t;
```

archaddr_t Interface type for native machine addresses.

The archaddr_t type is a 64-bit value, even on implementation architectures that use 32-bit addresses. This is done to avoid the need to duplicate the `coyotos.process.swapSpaceAndPC` methods and the corresponding field in the Process structure.

Wherever a kernel interface is specified as accepting an argument of type archaddr_t, the value passed must fall within the user-mode addressable bound of the underlying hardware architecture. Where the architecture specific annex specifies further constraints on valid user-mode addresses, the value passed must also satisfy those constraints.

```
typedef coyaddr_t archaddr_t;
```

payload_t Type declaration for protected payload values.

Issue: Not clear that this should live in the cap interface.

```
typedef uint32 payload_t;
```

exception_t Type to use when exception codes are passed as explicit arguments to methods.

```
typedef uint64 exception_t;
```

AllegedType Type declaration for the alleged interface type value.

```
typedef uint64 AllegedType;
```

Exceptions

OK No error occurred.

This ”exceptional result” supports the process teardown protocol, where the exiting process is expected to specify an exception code to return, and needs a way to say that no exception occurred. This exception code should

only be used when an exception code is explicitly passed as an argument value. It should *not* be used to indicate a non-exceptional result from a normally returning method.

InvalidCap Invoked capability was invalid.

Exceptional result returned when a capability has become invalid by virtue of its target object being destroyed.

Open Issue: there is a suggestion on the table that invoking an invalid capability should be viewed as an instruction execution exception rather than an invocation exception. I am provisionally inclined to the view that we should continue the (never implemented) EROS design in which an invocation exception can be conditionally propagate to the fault handler.

UnknownRequest Requested operation not implemented by this capability.

Exceptional result returned when the operation requested on a capability is not recognized by the implementing interface.

RequestError Message was malformed.

Exceptional result returned when the payload of a request does not correspond to the expected argument payload, or a provided argument falls outside the expected range.

When a operation is requested with insufficient permission using a malformed request, it is unspecified whether the `RequestError` or `NoAccess` exceptions is returned. `CapIdl`-generated services perform early argument demultiplexing, and therefore tend to generate the `RequestError` exception before considering `NoAccess` exception. Correctly written programs should not rely on this ordering preference, which may change at any time without notice.

NoAccess Insufficient permissions for requested operation.

Exceptional result returned when the operation is recognized but the operation requested requires permissions that are not conveyed by the invoked capability.

Operations

destroy Destroy the object.

```
void destroy();
```

The destroy operation requests that the target object destroy itself. This operation is not implemented by most kernel capabilities, but is declared as part of the basic capability interface because we want to establish a generally shared convention about the operation code used for this operation by those interfaces that actually implement it.

getType Get alleged type code.

```
AllegedType getType();
```

Returns an integral value indicating the alleged type code of the invoked interface.

coyotos.CapBits

Interface coyotos.CapBits

Derivation:

coyotos.Cap
coyotos.CapBits

Synopsis: Kernel interface to key representation.

KeyBits provides a means to inspect a capability as a value.

Structures

info Capability as data response structure.

```
struct info{
  uint32    w[4];
};
```

Operations

get
```
void get(
    Cap c,
    OUT info bits);
```

Return the representation of a key as data.

coyotos.CapPage

Interface coyotos.CapPage

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.AddressSpace
      coyotos.CapPage
```

Synopsis: Capability page interface

coyotos.Checkpoint

Interface `coyotos.Checkpoint`

Derivation:

```
coyotos.Cap
  coyotos.Checkpoint
```

Synopsis: Checkpoint control capability

The kernel cycles between normal execution and background checkpoint writeback. When executing normally, a snapshot can be declared, transitioning the kernel into the checkpoint aging state.

Exceptions

CkptIncomplete An attempt was made to initiate a new snapshot before writeback of the previous snapshot was completed.

Operations

snapshot Declare a new checkpoint.

```
void snapshot();
```

Raises: `CkptIncomplete`

processCheckpoint Make some implementation-defined amount of progress driving the pageout of the current snapshot. Return true if the checkpoint process requires more effort at the end of the current request. Return false if no further progress is required.

```
bool processCheckpoint();
```

coyotos.DevRangeCtl

Interface coyotos.DevRangeCtl

Derivation:

```
coyotos.Cap
  coyotos.DevRangeCtl
```

Synopsis: Authority to create and destroy device ranges.

The DevRangeCtl capability provides the authority to define and destroy device memory ranges.

Type Definitions

physaddr_t Type of a physical address.

```
typedef uint64 physaddr_t;
```

DevRangeNdx `typedef int32 DevRangeNdx;`

Operations

getNumRanges Return the number of device range structures implemented by the kernel in its present configuration.

```
DevRangeNdx getNumRanges();
```

getPagesPerRange Return the number of pages supported per range.

```
uint32 getPagesPerRange();
```

setup Define a new device memory range to the kernel.

```
void setup(
    DevRangeNdx range,
    physaddr_t base,
    physaddr_t bound,
    Memory.l2value_t l2sz);
```

Raises: RequestErrorNoAccess

Allocates a new kernel device range. Raises RequestError if l2sz is too small or if the requested device range overlaps some existing range. Raises NoAccess if the device range is in use or the requested range overlaps with the BSP's RAM ranges.

destroy Deallocate a device range and destroy all of its associated device pages.

```
void destroy(
    DevRangeNdx range);
```

coyotos.Discrim

Interface coyotos.Discrim

Derivation:

```
coyotos.Cap
  coyotos.Discrim
```

Synopsis: Discriminate among capability categories.

Enumerations

capClass Capability classifications returned by the classify operation.

Name	Type	Value
clNull	uint32	0
clWindow	uint32	1
clMemory	uint32	2
clSched	uint32	3
clEndpoint	uint32	4
clEntry	uint32	5
clProcess	uint32	6
clAppNotice	uint32	7
clOther	uint32	255

Operations

classify Return the classification of the passed capability, which is one of the selections in the capClass enumeration.

```
capClass classify(
    Cap c);
```

isDiscreet Return true exactly if this capability is discreet.

```
bool isDiscreet(
    Cap c);
```

A discreet capability is one that (transitively) conveys no authority to mutate. These include CapBits, Discrim, Null, Window, and LocalWindow. These also include weak Page, CapPage, and GPT capabilities.

compare Compare two capabilities for (exact) identity.

```
bool compare(
    Cap c1,
    Cap c2);
```

coyotos.Endpoint

Interface coyotos.Endpoint

Derivation:

coyotos.Cap
  coyotos.Endpoint

Synopsis: Endpoint interface

An endpoint is a destination for a message. Endpoints are described in detail in the Coyotos Microkernel Specification.

Operations

setRecipient Set the recipient process p.

```
void setRecipient(
    coyotos.Process p);
```

Raised the RequestError exception if the inserted capability is not a Process capability or a Null capability.

setPayloadMatch Enable protected payload matching.

```
void setPayloadMatch();
```

Note that payload matching cannot be disabled once enabled. This ensures that Entry capabilities which become invalid as a result of a payload match failure remain invalid.

setEndpointID Set the endpoint identifier value to id.

```
void setEndpointID(
    uint64 id);
```

getEndpointID Fetch the endpoint identifier valued.

```
uint64 getEndpointID();
```

makeEntryCap Fabricate an entry capability to this endpoint.

```
Cap makeEntryCap(
    payload_t payload);
```

makeAppNotifier Fabricate an application notify issuance capability to this endpoint capable of sending application-defined notice *n* exactly if bit *n* of allowedNotices is non-zero.

```
AppNotice makeAppNotifier(
    uint32 allowedNotices);
```

coyotos.GPT

Interface coyotos.GPT

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.AddressSpace
      coyotos.GPT
```

Synopsis: Guarded Page Table

A guarded page table (GPT) is the mechanism for composing address spaces. The data structure and basic idea of GPTs is discussed in the Coyotos Microkernel Specification. The mappable unit depends on the address space type:

Address Space Type	Mappable Unit
Data Space	bytes
Capability Space	capabilities
I/O Space	I/O ports

Constants

Name	Type	Value	Description
nSlots	l2value_t	16	number of slots in a GPT
l2slots	l2value_t	4	log(2) of number of slots in a GPT
handlerSlot	slot_t	15	
backgroundSlot	slot_t	14	

Operations

setl2v set the l2v field of this GPT to l2v, returning the previous l2v value.

```
l2value_t setl2v(
    l2value_t l2v);
```

getl2v Return the current l2v field value of this GPT.

```
l2value_t getl2v();
```

setHandler Set whether or not this GPT has a handler

```
void setHandler(
    bool hasHandler);
```

getHandler Find out whether this GPT has a handler

```
bool getHandler();
```

makeLocalWindow Store a new local window capability in slot `slot`.

```
void makeLocalWindow(
    slot_t slot,
    uint32 restr,
    coyaddr_t offset,
    guard_t guard,
    slot_t localRoot);
```

Raises: `NoAccessRequestError`

The `offset` argument specifies the offset *relative to the windowed space* named by the `localRoot` slot. The `restr` argument gives the permission restrictions imposed by the local window. The `guard` argument specifies the guard value for the window capability.

The supplied offset `offset` must be an integral multiple of 2^{l2v}. If this constraint is violated, the `RequestError` exception is raised.

makeBackgroundWindow Store a new background window capability in slot `slot`.

```
void makeBackgroundWindow(
    slot_t slot,
    uint32 restr,
    coyaddr_t offset,
    guard_t guard);
```

Raises: `NoAccessRequestError`

The `offset` argument specifies the offset *relative to the background space* that this window names. The `restr` argument gives the permission restrictions imposed by the window. The `guard` argument specifies the guard value for the new window capability.

The supplied offset `offset` must be an integral multiple of 2^{l2v}. If this constraint is violated, the `RequestError` exception is raised.

coyotos.IrqCtl

Interface coyotos.IrqCtl

Derivation:

coyotos.Cap
coyotos.IrqCtl

Synopsis: Low-level interrupt control capability.

An IrqCtl capability provides the authority to allocate IrqWait capabilities. wait on one of a range of interrupts. The invoker calls waitForInterrupt, which returns when the requested interrupt becomes in service. The IrqRange capability describes a range of interrupts that the wielder can wait for. A given process can wait for only one interrupt at a time.

Type Definitions

irq_t `typedef uint32 irq_t;`

Operations

getIrqWait Retrieve a capability enabling the wielder to wait for an interrupt.

```
IrqWait getIrqWait(
    irq_t irq);
```

bindIrq Mark an interrupt line as being bound to real hardware.

```
void bindIrq(
    irq_t irq);
```

wait Wait for an interrupt to occur on a given hardware interrupt line.

```
void wait(
    irq_t irq);
```

coyotos.IrqWait

Interface coyotos.IrqWait

Derivation:

coyotos.Cap
coyotos.IrqWait

Synopsis: Low-level interrupt control capability.

An IrqCtl capability provides the authority to allocate IrqWait capabilities. wait on one of a range of interrupts. The invoker calls wait(), which returns when the requested interrupt becomes in service. The IrqRange capability describes a range of interrupts that the wielder can wait for. A given process can wait for only one interrupt at a time.

Operations

wait Wait for an interrupt to occur on a given hardware interrupt line.

```
void wait();
```

coyotos.KernLog

Interface `coyotos.KernLog`

Derivation:

```
coyotos.Cap
  coyotos.KernLog
```

Synopsis: Capability to read/write the kernel log data.

Type Definitions

logString `typedef anon2 logString;`

Operations

log
```
void log(
        logString msg);
```

coyotos.LocalWindow

Interface `coyotos.LocalWindow`

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.Window
      coyotos.LocalWindow
```

Synopsis: Address space local window interface

Operations

getRootSlot Retrieve the `rootSlot` value stored in this window capability.

```
uint32 getRootSlot();
```

coyotos.Memory

Abstract Interface `coyotos.Memory`

Derivation:

```
coyotos.Cap
  coyotos.Memory
```

Synopsis: Operations common to all Coyotos memory-related capabilities.

The memory interface captures constant values and operations that are common to all memory-related capabilities.

Type Definitions

l2value_t Size of an address length.

```
typedef uint32 l2value_t;
```

guard_t Structure describing a guard word.

Ideally we would like to specify this as a bitfield, but the differences in bitfield packing rules from one platform to the next involve too many variations for `CapIDL` to handle. We therefore resort (reluctantly) to using a single word. The least significant 7 bits of the word contain the l2g value. The most significant 24 bits contain the match value.

```
typedef uint32 guard_t;
```

Enumerations

restrictions Values used in the memory capability permissions mask.

Issue: These values could be pre-biased to match the positioning of the type field. Should they be? How confident are we about the commitment to a 5-bit type field?

Name	Type	Value	Description
weak	uint32	1	Capability is weak. All capabilities fetched through a weak capability are returned with (conservatively) read only and weak permissions. If the kernel cannot determine how to perform this downgrade, the returned capability will be null.

Name	Type	Value	Description
readOnly	uint32	2	Capability is read only. This capability does not permit mutation of the target object.
noExecute	uint32	4	Capability does not permit execution. On hardware that supports a non-execute control bit, attempts to execute from a range marked `noExecute` will generate exceptions.
opaque	uint32	8	Capability is opaque. `GPT` does not permit slot fetch or store operations. This restriction is not meaningful for Page or `CapPage` objects.
noCall	uint32	16	Capability is no-call. No keeper will be called below this point in the memory traversal. This restriction is not meaningful for Page or `CapPage` objects.

Operations

reduce Return copy of current memory capability with reduced permissions.

```
Memory reduce(
    restrictions mask);
```

The returned capability will implement the same concrete interface as the invoked capability with appropriately reduced permissions.

Raises `RequestError` if an attempt is made to set the opaque or no-call bits on a Page or `CapPage` capability.

setGuard Return a capability having a different guard value.

```
Memory setGuard(
    guard_t guard);
```

The `guard` argument specifies the guard value to be set.

getGuard Return the guard value stored in the capability.

```
void getGuard(
    OUT guard_t guard);
```

getRestrictions Return the restriction bits set for this memory capability.

```
restrictions getRestrictions();
```

coyotos.MemoryHandler

Interface coyotos.MemoryHandler

Derivation:

coyotos.Cap
  coyotos.MemoryHandler

Synopsis: Kernel interface for per-process fault handlers.

A fault handler should implement this interface on its entry point capability. When delivering a fault to an external handler, the kernel will upcall the fault according to the protocol of this interface.

Operations

handle Kernel-invoked entry point for process fault handler.

```
/* oneway */ void handle(
    Process proc,
    Process.FC faultCode,
    uint64 faultInfo);
```

This entry point is invoked by the kernel. The capability `proc` names the process that incurred the fault. The `fault` structure provides the fault code and the issued address *relative to the managed `GPT`* at which the fault occurred.

Whether the fault will be resolved is at the discretion of the invoked handler. If the handler chooses to resolve the fault, it should take any action necessary to prevent recurrence and then call the `Process.resume` method on `proc` with the `clearFault` parameter set to `true.` This will clear the fault and allow the victim process to restart the faulting instruction.

If the memory fault handler is unable to resolve the fault, it should call the `Process.resume` method on `proc` with the `clearFault` parameter set to `false.` This will cause the process to restart with the fault still pending, which will cause the per-process handler to be invoked.

Note that the process capability `proc` supplied by the kernel at this interface is the restricted ”restart” form. It permits only the resume() method and the `getType`() method. Any other operation will generate a `NoAccess` response.

Note that this is a oneway invocation; the capability supplied in the reply slot at the time of upcall will be the Null capability. Merely replying in the conventional way is *not* sufficient to restart the faulted process.

coyotos.Null

Interface coyotos.Null

Derivation:

coyotos.Cap
  coyotos.Null

Synopsis: Universal invalid capability.

Capabilities to objects that have been destroyed become null capabilities. The null capability implements no operations of its own. It will respond to the getType() operation (only).

coyotos.ObStore

Interface coyotos.ObStore

Derivation:

```
coyotos.Cap
  coyotos.ObStore
```

Synopsis: Object storage manager interface

THIS IS A PLACEHOLDER

The object backing store manager responds to kernel-initiated requests for object pagein or object pageout. Like the cap.fault interface it is a "reverse" interface in the sense that it is kernel defined but server implemented.

The details of the object backing store interface are not yet determined.

coyotos.Page

Interface coyotos.Page

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.AddressSpace
      coyotos.Page
```

Synopsis: Page interface

coyotos.Process

Abstract Interface coyotos.Process

Derivation:

coyotos.Cap
coyotos.Process

Synopsis: Operations common to all Coyotos processes.

This is the architecture-independent process interface. For many operations of interest the architecture dependent interface should be consulted.

Type Definitions

capRegister `typedef uint32 capRegister;`

Enumerations

arch Process execution model (architecture).

This is primarily useful on machines that support (directly or through simulation) mutiple execution models. It allows a debugger to learn the execution model (architecture) of a subject process.

Name	Type	Value	Description
ia32	uint32	0	Intel IA32 (and derivatives)
coldfireV4	uint32	1	Later coldfire processors
arm	uint32	2	Acorn RISC Machine
amd64	uint32	3	AMD64
sparc	uint32	4	Sun 32-bit SPARC
sparc64	uint32	5	Sun 64-bit SPARC
ia64	uint32	6	Intel ITANIC

FC Fault (exception) codes.

The fault code provides a mostly machine independent renaming of the exception codes returned by the processor, plus a small number of additional exceptions generated by kernel software. Some architectures (notably IA32) extend this code space with additional, architecture-specific extensions. FC code points beginning at 128 are reserved for architecture-specific fault codes.

Name	Type	Value	Description
NoFault	uint8	0	Process currently does not have any fault.
MalformedSyscall	uint8	1	System call parameters were malformed, or system call number unknown.

Name	Type	Value	Description
SoftNotice	uint8	2	An application-defined interrupt is being delivered.
SliceExpired	uint8	3	Slice expiration occurred.
InvalidDataReference	uint8	4	Issued data reference address was undefined.
InvalidCapReference	uint8	5	Issued capability reference address was undefined.
NoExecute	uint8	6	Insufficient access rights for instruction reference. Note that this exception will be generated only on machines that provide the `NX` permission bit or equivalent functionality.
AccessViolation	uint8	7	Cannot write to specified address.
DataAccessTypeError	uint8	8	A data load/store was attempted to a capability page.
CapAccessTypeError	uint8	9	A capability load/store was attempted to a data page or I/O page.
MisalignedReference	uint8	10	Reference to item at misaligned address. Note that this is not considered a memory error.
TraverseLimit	uint8	12	`GPT` traversal limit exceeded in address space reference.
MalformedSpace	uint8	13	Address space `GPT` arrangement is malformed, or `GPT` contains an inappropriate capability type.
Notify	uint8	24	Issued to ensure notification delivery to in-process fault handler.
Startup	uint8	25	Issued when a process is first made running to allow activation handler to initialize.
NoAddrSpace	uint8	32	Process has invalid/maltyped address space capability. **Bug**: Not sure this can ever be issued – won't this simply manifest as one of `DataInvalidAddr`, `CapInvalidAddr`, or `IOInvalidAddr`?
NoSchedule	uint8	33	Process has invalid/maltyped schedule capability.
BreakPoint	uint8	34	Process encountered a breakpoint instruction (`PC`=&bpt). This fault code is used on architectures where the breakpoint instruction does not advance the program counter, or when the kernel can automatically roll the program counter back to point to the breakpoint instruction.
BrokePoint	uint8	35	Process encountered a breakpoint instruction (`PC`=&bpt+1). This fault code is used on architectures where it is impossible to automatically recover the correct address of the breakpoint instruction. On architectures where the address of the breakpoint instruction can be reliably re-established in software, the kernel will back up the instruction pointer and report the `BreakPoint` exception instead.

Name	Type	Value	Description
BadOpcode	uint8	36	Process issued an illegal or unknown instruction.
Alien	uint8	37	Process marked as "alien" performed an invocation instruction.
DivZero	uint8	38	Process performed an integer divide by zero
BadAlign	uint8	39	Process performed a checked misaligned memory reference.
NoFPU	uint8	40	No floating point unit available. This exception indicates that there is neither a hardware floating point unit nor a kernel-provided software emulation available on this machine.
FPfault	uint8	41	Unsupported floating point instruction. Other floating point error. More detailed information about the error should be obtained by executing the architecture specific `GetRegsFP`() operation and examining the appropriate floating point status register.
Debug	uint8	42	Debug Exception A hardware debugging event has occurred.
Overflow	uint8	44	Overflow trap.
Bounds	uint8	45	Bounds Violation.
SysCallEntry	uint8	46	Trap on Start of Invocation
SysCallReturn	uint8	47	Trap on Post-Receive
ActivationFail	uint8	48	Trap on Post-Receive
ObjectContentLost	uint8	49	Backing store has unrecoverably lost the state of this object.
GeneralProtection	uint8	128	General Protection fault.
StackSeg	uint8	129	Stack Segment fault.
SegNotPresent	uint8	130	Segment Not Present fault.
SIMDfp	uint8	131	`SIMD` floating point error.

cslot Capability slots of a process.

This is obsolete

Name	Type	Value	Description
handler	uint32	0	Fault handler.
addrSpace	uint32	1	Address space.
schedule	uint32	2	Schedule.
ioSpace	uint32	3	`IO` address space
cohort	uint32	4	Cohort capability.

Operations

resume Resume the process.

```
void resume(
    bool cancelFault);
```

Raises: `NoAccess`

Transitions the process to the ready state. When ready, a process will initiate new instructions according to its schedule, and may make progress in an invocation. If `cancelFault` is true, resume the process with its fault code set to `FC.NoFault`.

setSpaceAndPC Sets the address space and `PC` value as a single operation, and transitions the target process to the running state.

```
void setSpaceAndPC(
    Cap newAS,
    archaddr_t newPC);
```

Atomically installs newAS in the address space slot and newPC in the program counter slot. Typically used by a process on itself to switch to/from protospace.

If the operation is performed on the invoker, the load multiple caps and send phase are marked completed in the invoker register set. If the operation is performed on the invokee, these phases will already have completed.

Note that if this operation is performed on the invoker, it effectively causes the system call to be aborted after a successful kernel operation but without a receive or store multiple capabilities phase during the current kernel invocation. If this operation is performed on the invokee, the invokee is logically made running immediately, with the effect that the invokee receive phase is similarly terminated and no store multiple capabilities phase occurs during the current kernel invocation.

The phrase ”during the current kernel invocation” addresses a peculiar corner case. If the target process PC and AS point to the system call trap instruction prior to invocation, and the result of setSpaceAndPC is to set them to their prior values, the target process will resume by re-executing the system call trap. Depending on how much progress the target process has made during its system call, it may or may not then complete the system call by performing a receive phase and/or a store multiple capabilities phase. Note, however, that these actions will occur in the *subsequent* invocation.

getState Retrieve architecture-neutral state.

```
void getState(
    OUT FC faultCode,
    OUT archaddr_t faultInfo);
```

This interface is subject to change; at some point, runState and notices will be added, and everything will move into a structure.

setState Set architecture-neutral state.

```
void setState(
    FC faultCode,
    archaddr_t faultInfo);
```

This interface will fail with RequestError if faultCode is FC_NoFault and faultInfo is non-zero.

This interface is subject to change; at some point, runState and notices will be added, and everything will move into a structure. (note that this interface will **never** be able to set runState)

getSlot Retrieve the capability stored in the specified capability slot.

```
Cap getSlot(
    cslot slot);
```

setSlot Store the supplied capability stored to the specified capability slot. If the type of the stored capability is unsuitable for the slot, the RequestError exception is raised.

```
void setSlot(
    cslot slot,
    Cap c);
```

Raises: RequestError

getCapReg Retrieve the capability stored in the specified capability register.

```
Cap getCapReg(
    capRegister reg);
```

Raises: RequestError

If reg is out of bounds (>= 32), the RequestError exception is raised.

setCapReg Store the supplied capability stored to the specified capability register.

```
void setCapReg(
    capRegister reg,
    Cap c);
```

Raises: `RequestError`

If `reg` is out of bounds (>= 32), the `RequestError` exception is raised. If `reg` is zero, the `RequestError` exception is raised.

identifyEntryWithBrand Identify whether the passed entry capability `ent` is an entry capability to a process whose brand matches `brand.`

```
bool identifyEntryWithBrand(
    Cap ent,
    Cap brand,
    OUT payload_t pp,
    OUT uint64 epID,
    OUT bool isMe);
```

Returns true if `ent` is an entry capability, and the `brand` slot of the process designated (indirectly) by the entry capability matches the argument capability `brand.` If so, then the endpoint `ID` value of the endpoint named by `ent` is returned in `epID` and the protected payload value of `ent` is returned in `pp.` Otherwise, returns false, `epID` and `pp` are zero, and `isMe` is false.

If the endpoint names the invoked process capability, `isMe` will be true.

identifyEntry Identify whether the passed entry capability `ent` is an entry capability to a process whose brand matches ours.

```
bool identifyEntry(
    Cap ent,
    OUT payload_t pp,
    OUT uint64 epID,
    OUT bool isMe);
```

Functional specification is identical to `Process.identifyEntryWithBrand`, except that the brand used for comparison will be taken from the brand slot of the process designated by the process capability that was invoked.

amplifyCohortEntry Identify whether the passed entry capability `ent` is an entry capability to a peer thread.

```
bool amplifyCohortEntry(
    Cap ent,
    OUT payload_t pp,
    OUT uint64 epID,
    OUT Process peer);
```

Return true if `ent` is an entry capability, and the `cohort` slot of the process designated (indirectly) by the entry capability matches the `cohort` slot of the invoked process.

If the capability `ent` *does* name a peer process, returns the endpoint `ID` value in `epID,` the entry capability's protected payload value in `pp,` and the peer process capability in `peer.`

coyotos.ProcessHandler

Interface coyotos.ProcessHandler

Derivation:

coyotos.Cap
  coyotos.ProcessHandler

Synopsis: Kernel interface for per-process fault handlers.

A fault handler should implement this interface on its entry point capability. When delivering a fault to an external handler, the kernel will upcall the fault according to the protocol of this interface.

Operations

handle Kernel-invoked entry point for process fault handler.

```
/* oneway */ void handle(
    Process proc,
    Process.FC faultCode,
    uint64 faultInfo);
```

This entry point is invoked by the kernel. The capability `proc` names the process that incurred the fault. The `fault` structure provides the fault code and the issued address at which the fault occurred.

Whether the fault will be resolved is at the discretion of the invoked handler. If the handler chooses to resolve the fault, it should take any action necessary to prevent recurrence and then call the `Process.resume` method with the `clearFault` parameter set to true.

Note that in contrast to `MemoryHandler.handle`, the process capability provided here is a full-strength capability.

Note that this is a oneway method; no reply is expected. The capability supplied in the reply slot at the time of upcall will be the Null capability. Merely replying in the conventional way is *not* sufficient to restart the faulted process.

coyotos.Range

Interface coyotos.Range

Derivation:

coyotos.Cap
coyotos.Range

Synopsis: Kernel interface to object ranges.

The Range interface provides the primitive means for the allocation and deallocation of object capabilities.

All allocation operations return objects whose data fields are set to zero and whose capability fields are set to cap.null.

Constants

Name	Type	Value	Description
devOidStart	oid_t	18374686479671623680	Beginning of physical OID frames. OIDs in the range [0xff00000000000000ull, 0xffffffffffffffffull] are reserved for physical object frame allocations associated with device memory. Allocating one of these OIDs has the effect of allocating the physical object frame whose frame number *n* is (oid-0xff00000000000000ull), provided that a corresponding object frame of the requested type exists within the appropriate kernel object vector.

Type Definitions

oid_t Type of an object ID.

```
typedef uint64 oid_t;
```

Exceptions

RangeErr Attempted to allocate object from unbacked/undefined range.

NotMounted Attempt was made to allocate an object from a dismounted range.

IoErr An I/O error occurred during object allocation.

AllocCountRollover Requested object could not be allocated because the allocation count has reached its maximal value.

Enumerations

obType Object types returned by identify.

This must match the enumeration in obstore/`ObFrameType.h`

Bug: At some point the enumeration here should become definitive, and the enumeration in `ObFrameType.h` should be retired.

Name	Type	Value
otPage	uint32	0
otCapPage	uint32	1
otGPT	uint32	2
otProcess	uint32	3
otEndpoint	uint32	4
otNUM_TYPES	uint32	5
otInvalid	uint32	4294967295

Operations

identify Identify an object.

```
void identify(
    Cap c,
    OUT obType type,
    OUT uint64 offset);
```

Determines the `OID` and type of an object. If the capability `c` is a non-object capability, he reported `type` will be `otInvalid` and the reported `offset` will be 0.

rescind Rescind an object.

```
void rescind(
    Cap c);
```

Rescinds the passed capability if it is the "full powered" variant of this capability type. That is: a read-write page, cappage, or gpt capability, a process capability, an endpoint capability, or a rcvqueue capability, but not an enty capability or any reduced form of page/cappage capability.

nextBackedSubrange Report next populated subrange.

```
void nextBackedSubrange(
    oid_t startOffset,
    obType type,
    OUT oid_t base,
    OUT oid_t bound);
```

Returns the offset and length of then next subrange of type `type` for which backing objects actually exist, starting the search at offset /`startOffset`/ of the passed range key.

Note: This operation may be obsoleted in future versions of the interface, as the idea of `OID` to location correspondence may away.

getCap Create capability to object.

```
Cap getCap(
    oid_t oid,
    obType ty);
```

Raises: `RangeErrNotMountedAllocCountRollover`

Returns object capability to an object of the specified type `ty` having the given `oid.` If the range is dismounted or otherwise unavailable, this operation will throw an exception.

waitCap Create capability to object (blocking).

```
Cap waitCap(
    oid_t oid,
    obType ty);
```

Raises: `AllocCountRollover`

Returns an object capability to an object of the specified type `ty` having the given `oid.` If the range is dismounted or otherwise unavailable, this operation will block until it becomes available.

getProcess Create capability to process.

```
Process getProcess(
    oid_t oid,
    Cap brand);
```

Raises: `RangeErrNotMountedAllocCountRollover`

Creates a process capability having the specified `oid.` The process brand slot will be initialized to the value of `brand.` If the range is dismounted or otherwise unavailable, this operation will throw the `RangeErr` exception.

waitProcess Create capability to process (blocking).

```
Process waitProcess(
    oid_t oid,
    Cap brand);
```

Raises: `AllocCountRollover`

Creates a process capability having the specified `oid.` The process brand slot will be initialized to the value of `brand.` If the range is dismounted or otherwise unavailable, this operation will block until it becomes available.

coyotos.RcvQueue

Interface coyotos.RcvQueue

Derivation:

coyotos.Cap
coyotos.RcvQueue

Synopsis: Receive queue interface.

This interface is a place-holder for a future implementation that may never happen. This interface may be dropped in future specification revisions.

Operations

dequeue Dequeue an endpoint from this receive queue if one is present, and return a capability to it via ep. Return value is true if an endpoint has been returned.

```
bool dequeue(
    OUT Endpoint ep);
```

makeSendCap Fabricate a sender's capability to this receive queue.

```
Cap makeSendCap(
    payload_t payload);
```

coyotos.SchedCtl

Interface coyotos.SchedCtl

Derivation:

coyotos.Cap
  coyotos.SchedCtl

Synopsis: Low-level scheduler control capability.

THIS IS A PLACEHOLDER

This will eventually be the interface that is used by the application-level admission control agent to introduce new entries into the kernel schedule table.

coyotos.Schedule

Interface `coyotos.Schedule`

Derivation:

```
coyotos.Cap
  coyotos.Schedule
```

Synopsis: Base scheduling interface.

THIS IS A PLACEHOLDER

This will eventually be the interface that is used by an application to define its scheduling criteria.

coyotos.Sleep

Interface coyotos.Sleep

Derivation:

coyotos.Cap
  coyotos.Sleep

Synopsis: Kernel delay mechanism.

The sleep capability allows a process to block for a specified period of time. Note that the sleep(), if any, happens before the reply. It is the invoker rather than the invokee who blocks.

In contrast to all other kernel interfaces, the sleep capability contrives to be in a non-receiving state until the requested interval has expired.

Operations

sleepTill Sleep until the specified number of seconds `sec` and microseconds `usec` have passed since the beginning of the current restart epoch.

```
void sleepTill(
    uint32 sec,
    uint32 usec);
```

Raises `RequestError` if `usec` $>= 10^6$

sleepFor Sleep until the specified number of seconds `sec` and microseconds `usec` have passed. Note that this will be re-written by the kernel into a `sleepTill` invocation.

```
void sleepFor(
    uint32 sec,
    uint32 usec);
```

coyotos.SysCtl

Interface `coyotos.SysCtl`

Derivation:

```
coyotos.Cap
  coyotos.SysCtl
```

Synopsis: System control capability

This capability provides the authority to perform low-level system control functions, notably halt, shutdown, and reboot.

Operations

halt Halt the system immediately. This is primarily useful for kernel debugging.

```
void halt();
```

powerdown Power the hardware down.

```
void powerdown();
```

reboot Reboot the hardware.

```
void reboot();
```

coyotos.Window

Interface coyotos.Window

Derivation:

```
coyotos.Cap
  coyotos.Memory
    coyotos.Window
```

Synopsis: Address space background window interface

Operations

getOffset Retrieve the offset value stored in this window capability.

```
coyaddr_t getOffset();
```

coyotos.coldfire.Process

Abstract Interface `coyotos.coldfire.Process`

Derivation:

```
coyotos.Cap
  coyotos.Process
    coyotos.coldfire.Process
```

Synopsis: Operations common to all Coyotos processes.

This is the architecture-independent process interface. For many operations of interest the architecture dependent interface should be consulted.

Structures

fixregs Architecture-specific fixed-point registers layout

```
struct fixregs{
  uint32   ExceptAddr;   Other address associated with exception, such as page
                         fault virtual address.
  uint32   d[8];
  uint32   a[8];
  uint32   ExceptWord;   First word of exception frame. This includes the con-
                         dition code register in its least byte.
  uint32   ExceptPC;     PC of instruction incurring exception.
                         This is usually the current program counter.
};
```

floatregs Architecture-specific floating point registers.

```
struct floatregs{
  uint32   fpcr;
  uint32   fpsr;
  uint32   fpiar;
  uint64   fp[8];
};
```

emacregs Extended multiply-accumulate unit registers.

```
struct emacregs{
  uint32   macsr;
  uint32   acc[4];
  uint32   accExt01;
  uint32   accExt23;
  uint32   mask;
};
```

Operations

getFixRegs Fetch the fixed-point register set.

```
fixregs getFixRegs();
```

setFixRegs Set the fixed-point register set.

```
void setFixRegs(
    fixregs regs);
```

The overwrite of the register set area occurs atomically.

A `RequestError` exception will be raised if the size of the provided register structure does not match the size of the register set being updated.

getFloatRegs Fetch the floating-point register set.

```
floatregs getFloatRegs();
```

setFloatRegs Set the floating-point register set.

```
void setFloatRegs(
    floatregs regs);
```

The overwrite of the register set area occurs atomically.

A `RequestError` exception will be raised if the size of the provided register structure does not match the size of the register set being updated.

getEmacRegs Fetch the `EMAC` unit register set.

```
emacregs getEmacRegs();
```

setEmacRegs Set the `EMAC` unit register set.

```
void setEmacRegs(
    emacregs regs);
```

The overwrite of the register set area occurs atomically.

A `RequestError` exception will be raised if the size of the provided register structure does not match the size of the register set being updated.

coyotos.i386.Process

Abstract Interface `coyotos.i386.Process`

Derivation:

```
coyotos.Cap
  coyotos.Process
    coyotos.i386.Process
```

Synopsis: Operations common to all Coyotos processes.

This is the architecture-independent process interface. For many operations of interest the architecture dependent interface should be consulted.

Type Definitions

sse_reg `typedef uint32 sse_reg[4];`

Structures

fixregs Architecture-specific fixed-point registers layout

```
struct fixregs{
  uint32  EDI;
  uint32  ESI;
  uint32  EBP;
  uint32  ExceptAddr;
  uint32  EBX;
  uint32  EDX;
  uint32  ECX;
  uint32  EAX;
  uint32  ExceptNo;
  uint32  Error;
  uint32  EIP;
  uint32  CS;
  uint32  EFLAGS;
  uint32  ESP;
  uint32  SS;
  uint32  ES;
  uint32  DS;
  uint32  FS;
  uint32  GS;
};
```

floatregs Architecture-specific floating point and `SIMD` registers layout.

```
struct floatregs{
    uint16    fcw;
    uint16    fsw;
    uint16    ftw;
    uint16    fop;
    uint32    eip;
    uint32    cs;
    uint32    dp;
    uint32    ds;
    uint32    mxcsr;
    uint32    mxcsr_mask;
    sse_reg   fpr[8];
    sse_reg   xmm[8];
};
```

There is a policy problem here: Intel changed the format starting in the Pentium-III (FSAVE -> FXSAVE), and they have reserved space for future extensions. This raises a tricky question about what to present at the interface.

It is possible to reconstruct the FSAVE information from the FXSAVE information. It is also possible to run on an older machine by initializing the SSE slots to well-defined values and pretending that since no SSE instructions were executed the state must not have changed. That is what we do here.

This leaves us with the question of what type to present at the interface. Our sense is that it is better to present a well-defined, shorter structure than to give a general structure whose internals will need to be edited later. If necessary we can add a new fetch operation to deal with future extensions to SSE state.

Operations

getFixRegs Fetch the fixed-point register set.

```
fixregs getFixRegs();
```

setFixRegs Set the fixed-point register set.

```
void setFixRegs(
    fixregs regs);
```

The overwrite of the register set area occurs atomically.

A RequestError exception will be raised if the size of the provided register structure does not match the size of the register set being updated.

getFloatRegs Fetch the floating-point and SIMD register set.

```
floatregs getFloatRegs();
```

setFloatRegs Set the floating-point and SIMD register set.

```
void setFloatRegs(
    floatregs regs);
```

The overwrite of the register set area occurs atomically.

A RequestError exception will be raised if the size of the provided register structure does not match the size of the register set being updated.

Part IV

Architecture Specific Annexes

Appendix A

IA-32 Interface

The kernel header file `coyotos/i386/UPCB.h` defines the UPCB layout for this architecture.

A.1 Execution Models

The IA-32 implementation supports the "small spaces" optimization. If a process restricts its address references (ignoring KIP references) to the inclusive range [0, 0x10000], the kernel will attempt to run it in a small space region. Control transfers between two applications running in a small address space, or between a large address space and a small address space, are significantly faster than large space control transfer.

By referencing an address outside of the small space bound, a process signals that it wishes to be treated as a large address space process. The user-mode addressable range of a large address space is [0x0,0xC0000000]. Because this transition is transparent to the process, the value returned by `LSL` (load segment limit) is subject to change between any two instructions. The transition between large and small address space models is otherwise transparent to application code.

A.2 System Call Trap Interface

In all cases the register utilization convention follows the requirements of SYSENTER:

Register	**Input**	**Output**
EAX	`IPR0`	`OPR0`
EBX	`IPR1`	`OPR1`
ECX	*Post-syscall SP*	*Undefined*
EDX	*Post-syscall return PC*	*Undefined*
ESI	`IPR2`	`OPR2`
EDI	`IPR3`	`OPR3`
EBP	`InvokeCap: invCap` `Other: unused`	`InvokeCap: rcvPP` `Other: unaltered`
ESP	*Unavailable*	*input ECX value*

Table A.1: System call entry and exit conventions.

Different generations of IA-32 implementations require different system call implementations for efficiency. Depending on the hardware implementation, either a software interrupt (`int $0x31`) the SYSENTER instruction, or the SYSCALL instruction may be used. The software interrupt entry point may be used on all platforms. The preferred

entry and exit mechanism is left to the kernel implementation. The kernel publishes the preferred mechanism via the **kernel interface page**. The system call trap instruction appears at offset zero of this page.

The kernel interface page is mapped into all user address spaces at a well-known *far* address: `0x32:0`.[1] The protocol for invoking the preferred system call trap instruction is to marshal all arguments and perform a *far jump* to this address. The use of a far address allows the kernel interface page to be accessible to both large and small address spaces within a single compilation and execution model. This address is a well-known constant that is part of the architecture specification, and will not change in future versions of Coyotos.

A.3 Virtual Registers

The architecture defines the following locations for input and output parameters, buffer registers, and capability invocation paramaters. Locations described with an identifier rather than a register name indicate a field in the UPCB structure.

Virtual Register	Location	Virtual Register	Location
IPR0	`%eax`	OPR0	`%eax`
IPR1	`%ebx`	OPR1	`%ebx`
IPR2	`%esi`	OPR2	`%esi`
IPR3	`%edi`	OPR3	`%edi`
IPR4..IPR7	*caller stack*	OPR4..OPR7	*receiver stack*
invCap	*caller stack*	rcvPP	`softregs.rcvPP`
rcvEpID	`softregs.epID`		

Values in the `softregs` structure are copied out to the receiver stack after the receive phase if IPR0.CO is set (1).

A.4 Thread Identification

This section is stale

The first two words of the UPCB are used by the application-level runtime system to provide support for multi-threading. Word 0 should be used by the multi-threading library to store the pointer to the thread control block. Word 1 should be used to store the virtual address of the UPCB itself, as seen by the application. Neither word is initialized by the kernel. Provided the address range exposed falls within the user-mode addressable range, the value of UPCB word 1 is loaded as the base address of the segment named by %GS, with a limit value of 0x1000.

The critical effect of this is that %GS:0 can be used to load and store the application-level thread control block pointer, which is in turn used to access thread-local storage [21].

Open Issue

While storing the UPCB VA in the UPCB is attractive, it is not necessarily efficient. The problem is that the value must be range checked on every kernel exit because it can be modified by user-mode code. It may be better to introduce a system call for this and store it in the Process instead.

A.5 Device Ranges

A minimum of 16 device ranges, each supporting 16 device pages, must be provided by the implementation.

[1] The selector value is provisional.

Part V

Notes on Implementation

The Coyotos specification may be seen as defining an abstract machine architecture. In a real-world implementation, this abstract machine must be mapped on to a combination of hardware and software by the kernel implementation. This mapping must satisfy two properties:

- **The permissions of the implementation state must never exceed those of the abstract state.** Any instruction whose effect is permitted by the implementation state must be permitted by the abstract machine state. The implementation state is a conservative approximation of the abstract state.
- **Ignoring latency, every instruction whose execution is permitted by the abstract machine must ultimately be permitted by the implementation.** The implementation state is an approximation of the abstract state that is constructed on demand.

  The means by which demand update is triggered are the system call trap and the various protection and permissions violation traps. This induces several requirements on the underlying hardware:

  – The hardware must implement page-granularity protections (or better) in its memory management unit.
  – The hardware must implement precise exceptions — or precise enough that the software implementation can correct them (e.g. the Pentium family's breakpoint trap incorrectly advances the program counter, but the amount of the advance is known and be corrected in software.
  – Preferred hardware must implement a "no-execute" permission. This is a recently rediscovered feature in the hardware world, and the specification does not require `NX` permissions to be enforced on hardware that does not provide this feature.

Any change to the abstract state that reduces permissions must be reflected by an immediate change in the hardware state that (conservatively) maintains these invariants. In some cases it is not obvious how to do this. This part of the specification discusses possible implementation techniques for several key parts of the dependency tracking implementation.

The entirety of this part is non-normative.

Appendix B

Implementation of Capabilities

Because so much of any implementation depends on the internal representation of capabilities, we begin with a brief discussion of capability representation choices.

It is convenient for capabilities to have two forms, which we call **prepared** and **unprepared**. The unprepared form is the one described in Section˜2.1. The `P` bit indicates whether the capability is prepared (1) or unprepared (0). The representation of a prepared capability is a matter that is private to the kernel. When the kernel discloses capability representation to application code, it *always* discloses the unprepared format.

B.1 Unprepared Capabilities

An unprepared capability may be valid or invalid, because it is not known from the capability whether the object it names has been destroyed subsequent to the creation of the capability. This can only be known by comparing the `allocCount` of the capability to the `allocCount` of the object itself.

Assuming the capability is valid, the object designated by an unprepared capability may or may not be in memory. This can only be determined by performing an object lookup to discover whether the object is in memory. In all current implementations of KeyKOS, EROS, and Coyotos, a hash table is maintained that provides a mapping from object id (`OID`) to the actual object for every object that is currently in memory. In systems supporting transparent persistence, there may be *two* objects for a given `OID`: the current one and the one that was current at the time of the last snapshot.

B.2 Prepared Capabilities — Linked Implementation

In KeyKOS and EROS, a prepared capability designates an object that is known to be in memory. The prepared capability points directly to this object. In addition, the prepared capability resides on a "key chain", which is a circularly linked list whose "head" is part of the object header. This allows the object to be efficiently found given the capability, and also allows all prepared capabilities to be found given the object.

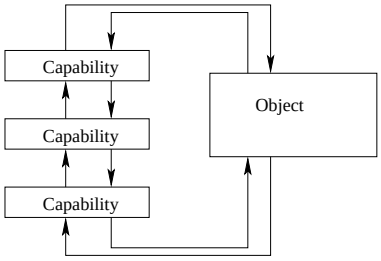

Figure B.1: EROS/KeyKOS key chain

There are several advantages to this design:

- Once a capability is determined to be prepared, the object is known to be in memory as a consequence of invariants, and few further checks related to residency or well-formedness are necessary. In addition, the capability is known to be valid, in the sense that its `allocCount` matches that of the target object.
- When an object is destroyed, it is straightforward to locate the active capabilities to the object and rewrite them as invalid capabilities.
- When an object is to be removed from memory, it is straightforward to locate all capabilities to the object and restore them to their unprepared form.
- In either the destruction or pageout case, it is straightforward to identify the *locations* of all prepared capabilities. KeyKOS and EROS exploit this property very aggressively. They maintain a significant amount of dependency information in hashed structures that are indexed by capability address. For example, the KeyKOS/EROS "depend table" is a hash table of (capability address, page table entry address) pairs.

However, there is a key disadvantage as well:

- Capability copy is a frequent operation. Empirically, we found in EROS that many copied capabilities were prepared, that most overwritten capabilities were prepared, and that updating the key chain when both the source and destination capabilities are prepared entails three cache misses per copy to access the neighbors (Figure 10.2). On modern machines, this cost is a substantial fraction of the total IPC cost.
- Capability invocation may take time O(n), where n is the size of memory. This is possible because an arbitrary number of resume capabilities may accumulate on the key chain, and the chain needs to be traversed in order to destroy them.

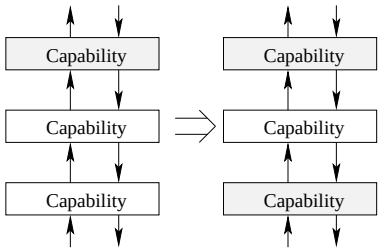

Figure B.2: EROS/KeyKOS key copy

B.3 Prepared Capabilities — Scavenged Implementation

In Coyotos, a prepared capability does *not* guarantee either that the target object is in memory or that the prepared capability is valid. We are accepting weaker invariants in order to improve capability copy performance. Instead of a linked list, the relationship between a capability and its object is shown in Figure 10.3.

In this design, the capability points to the object (equivalently: holds an index), but also contains a pointer (equivalently: an index) to an `ObTable` structure. The object pointer is valid only if the capability's `ObTable` reference matches the `ObTable` reference in the object itself. The purpose of the `ObTable` structure is to hold the information needed to deprepare the capability back to its on-disk form. This consists of a copy of the object ID (because that field of the capability is overwritten by the references) and a valid bit. Under normal circumstances, an object has one `ObTable` entry.[1]

[1] There may be none if there are no prepared capabilities to this object, but this is a transient case because object page-in is induced only by capability preparation.

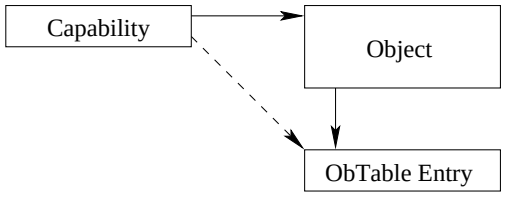

Figure B.3: Capability/object relationship

When an object is paged out, the first step is to change the `ObTable` reference in the object header to Null. This ensures that all outstanding capabilities will be deprepared back to their on-disk format. When an object is destroyed, the valid bit in the current `ObTable` entry is set to "invalid" before the `ObTable` reference in the object header is nullified. In this situation, outstanding capabilities will be deprepared to the Null capability. Capabilities are deprepared through a combination of proactive update when the containing object is written to store and background scavenging.

B.3.1 Scavenging

This design requires an oversupply of `ObTable` structures. `ObTable` structures are freed by background scavenging. The scavenging proceeds as follows:

1. Make a pass over all `ObTable` structures. Mark the ones that are current. Clear the mark on the ones that are not current.
2. Traverse all in-memory objects that contain capabilities. For each capability:
   - If the `ObTable` entry is current, leave the capability alone.
   - If the `ObTable` entry is invalid, deprepare the capability to null.
   - If the `ObTable` entry is valid but not current, deprepare the capability back to it's on-disk form.
3. Make a second pass over all `ObTable` structures, placing the unmarked structures onto the free list.

This algorithm can be implemented incrementally by forcing some progress whenever an `ObTable` entry is allocated.

Note that the algorithm is *not* trying to detect unreferenced objects. This is the responsibility of the aging logic, which is handled separately.

When the algorithm is executed incrementally, there is the usual race between the mark pass and the mutator that is copying and overwriting references. The unfortunate case is the sequence where:

1. The mark pass has moved past capability slot A, but has not yet reached slot B.
2. Slot B holds the only outstanding capability to some object.
3. A copy is made from slot B to slot A.
4. Slot B is overwritten before it is reached by the mark pass. A now holds the only capability to the `ObTable` entry, but it will not be seen by the mark pass.

Note that the race condition does not matter if B is a *valid* capability, because in this case the `ObTable` structure is already marked. The issue arises only when B is an *invalid* capability. The race is resolved by checking whenever a prepared capability is copied, and updating the newly written capability to the null capability if it is invalid.

B.3.2 Pros and Cons

The primary advantage of this design over the KeyKOS/EROS design is speed of capability copy. There is only one marginal cache miss per copy, which is the probe that checks the capability for validity. If it is feasible to scan all processes in a non-incremental fashion, this probe can be eliminated in the fast IPC path because any copy proceeding *from* a capability register does not need to be checked for validity. The need for this optimization should be guided by measurement, and our initial implementation will not attempt it.

The main disadvantage is that this design requires an incremental scavenging pass whose performance cost is not yet known.

Appendix C

Mapping Dependencies

The most challenging set of structures to design in the Coyotos implementation is the mechanism for keeping GPT structures, objects, and page table entries consistent. There are three requirements:

1. When a page is destroyed or removed from memory, all page table entries that point to that page must be invalidated.

2. When a GPT is destroyed or removed from memory, all currently valid translations in the page table structures that were constructed by traversing the GPT must be invalidated.

3. When a capability slot within a GPT is overwritten, all currently valid translations in the page table structures that were constructed by traversing that slot of the GPT must be invalidated. This may viewed as a sub-case of (2).

C.1 Page Removal

In KeyKOS and EROS, the key chain meant that any implementation of requirement (3) also satisfied requirement (1). When a page is removed from memory, its key chain can be traversed to locate all of the capability slots that reference the page. These can then be used to invalidate the necessary page table entries.

In the Coyotos implementation, which does not have a key chain, we maintain a reverse page table structure known as `PTE-1`. For every valid page table entry in the hardware page table, we maintain a reverse entry that provides a mapping from the physical object address to the kernel virtual address of its referencing page table entry.

In systems having hierarchical page tables, we maintain this inverse page table structure at all levels of the translation hierarchy. This allows mapping tables to be aged and reclaimed.

C.2 GPT Dependencies

The statement of requirements in (2, 3) is a bit subtle. The straightforward implementation of these requirements is to record a pairwise relationship between capability slot addresses and page table entry addresses, and use this to invalidate all page table entries when a slot is overwritten or a GPT is removed. This was, in essence, the implementation used by KeyKOS and EROS, and it is fairly straightforward to see why it satisfies the requirement that "the permissions of the implementation state must never exceed those of the abstract state." However, this implementation is both unnecessarily aggressive and unnecessarily expensive.

Because we do not rely on key rings for page removal, Coyotos has slightly different properties than KeyKOS or EROS. Whenever we consider changing the value of a slot, we always have the address of its containing GPT in

hand.[1]. Coyotos therefore maintains a dependency table that maps from GPT addresses to page table entry addresses. There may be multiple entries in this table for a given GPT. This can happen for two reasons:

1. A GPT may produce multiple page tables because it spans multiple entries in a page directory (Figure 11.1). This arises only in hardware system having hierarchical translation systems, but it can arise at any "layer" of the translation system. In fact, a single GPT can span three or more layers if it describes the only valid path across multiple levels of the hardware tables.

2. Because of page table reuse, a GPT may produce both read-only and read-write variants of the same page table.

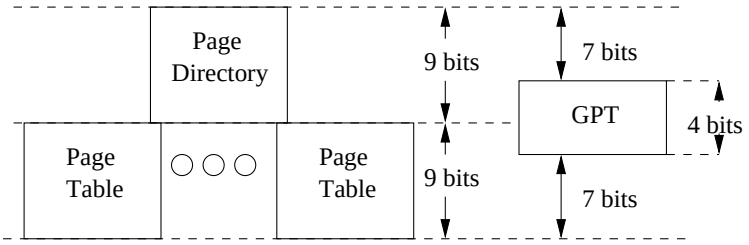

Figure C.1: Capability/object relationship

The hierarchical case is complicated by the desire for page table sharing. In KeyKOS and EROS, we recorded dependency information at all implicated levels of the hardware translation hierarchy. In Coyotos we do not. Instead, we record dependencies only when the GPT wholly or partially dominates the hardware table. This is sufficient to let is invalidate all *paths* through the hardware tree that are implicated by changes to a GPT. It does *not* allow us to invalidate all of the page table entries, but we assert this is not actually necessary to satisfy the requirements.

This assertion is *not* obvious and will need to be confirmed by demonstration as we develop a statement of invariants maintained by the translation logic in this case. The substance of it, however, is that valid entries higher in the hardware structures do not matter if all of the lower entries they span are correctly invalidated. If the GPT is being destroyed, the lower tables will in due course be reclaimed and the higher-level page table entries will then be invalidated. If the GPT is being overwritten, the higher-level page table entries would get rebuilt in any case, and the permissions of the lower-level page table entries are sufficient to ensure a conservative mapping of the abstract machine's permission state.

C.3 Optimizations

Ironically, the new structure should be more compact than the old structure.

Inverse Page Table The `PTE-1` table requires only a single word per entry, because the pointer to the PTE can be used to read the physical page address in order to detect hash collisions and stale dependency table entries.

Observe further that the majority of pages and page tables have only one or two simultaneous page table entries. A possible storage optimization is to dedicate two `PTE-1` entries in the frame management structures for each of these, leaving the general `PTE-1` table to handle only those pages that are widely shared. Whether this is worthwhile depends on whether a sufficiently good hash function can be discovered to keep hash chains short in the usual case.

GPT Dependencies Because we use GPT object pointers rather than GPT slot pointers in the GPT dependency table, our GPT dependency table will have a factor of 16 (well, given underutilization probably a factor of 5 to 8) reduction in space requirements compared to the old dependency tracking scheme.

Observe that while a GPT may produce entries in multiple page tables, it always produces 2^k entries at a natural alignment boundary within those tables. This statement is also true of GPT *slots*, and offered a basis for run length

[1] In EROS, we knew the location of the vector of Nodes (the precursor to GPTs), and we could use this knowledge to infer the containing Node address from any given Node slot address. This inference is not memory safe, and we wanted to avoid it in Coyotos in anticipation of later verification efforts.

compression of the GPT dependency table in some implementations. We note that this option remains available in the new implementation.

Bibliography

[1] Kevin John Elphinstone. *Virtual Memory in a 64-Bit Microkernel*. Ph.D. Dissertation. University of New South Wales, School of Computer Science, August 31, 1999.

[2] Norman Hardy. "The KeyKOS Architecture." *Operating Systems Review*, **19**(4), October 1985, pp. 8–25.

[3] Jochen Liedtke and Kevin Elphinstone. *Guarded Page Tables on the MIPS R4600, or, An Exercise in Architecture-Dependent Micro Optimization*. Technical Report UNSWCSE-TR-9503, University of New South Wales, School of Computer Science, 1995.

[4] J. S. Shapiro, J. M. Smith, and D. J. Farber. "EROS, A Fast Capability System" *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999. pp. 170–185. Kiawah Island Resort, SC, USA.

[5] J. S. Shapiro, J. Adams. "Design Evolution of the EROS Single-Level Store" *Proc. 2002 USENIX Annual Technical Conference*. 2002. pp. 59–72.

[6] J. Shapiro, M. Doerrie, S. Sridhar, M. Miller. "Towards a Verified, General-Purpose Operating System Kernel" *Proc. NICTA OS Verification Workshop 2004*. October, 2004. Sydney, New South Wales, Australia.

[7] J. S. Shapiro and S. Weber. "Verifying the EROS Confinement Mechanism." *Proc. 2000 IEEE Symposium on Security and Privacy*. May 2000. pp. 166–176. Oakland, CA, USA

[8] J. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. "Design of the EROS Trusted Window System" *Proc. 13th USENIX Security Symposium*. 2004

[9] —: L4 *eXperimental* Kernel Reference Manual. System Architecture Group, Dept. of Computer Science, Universität Karlsruhe. 2004

[10] A. Sinha, S. Sarat, and J. S. Shapiro. "Network Subsystems Reloaded" *Proc. 2004 USENIX Annual Technical Conference*. Dec. 2004

[11] J. B. Dennis and E. C. van Horn. "Programming Semantics for Multiprogrammed Computations" *Communications of the ACM*. **9**(3), March 1966. pp. 143–154.

[12] J. Liedtke. "Improving IPC by Kernel Design" *Proc. 14th ACM Symposium on Operating System Principles.* ACM. pp. 175–188. 1993

[13] NOT USED J. S. Shapiro, D. J. Farber, and J. M. Smith. "The Measured Performance of a Fast Local IPC" *Proc. 5th International Workshop on Object Orientation in Operating Systems*. Seattle, WA, USA. Nov 1996. pp. 89–94. IEEE.

[14] NOT USED B. Ford and J. Lepreau. "Evolving Mach 3.0 to a Migrating Threads Model" *Proc. 1994 Winter USENIX Conference*. Jan 1994. pp. 97–114.

[15] NOT USED T. Roscoe. *The Structure of a Multi-Service Operating System*. Ph.D. Dissertation, University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR376. August 1995.

[16] W. A. Wulf, E. S. Cohen, W. M. Corwin, A. K. Jones, R. Levin, C. Pierson and Fred J. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System" *Communications of the ACM*. **17**(6), pp. 337–345. 1974.

[17] D. D. Redell. *Naming and Protection in Extensible Operating Systems.* Ph.D. Dissertation. Department of Computer Science, University of California at Berkeley. Nov 1974.

[18] S. A. Rajunas. *The KeyKOS/KeySAFE System Design.* Key Logic Technical Report SEC009-01. March 1989. Key Logic, Inc.

[19] B. Kauer. *L4.sec Implementation — Kernel Memory Management* Diploma Thesis, Chair for Operating Systems, Technical University of Dresden. Supervisor: Marcus Volp. 2005

[20] Motorola, Inc. *MC68851 Paged Memory Management Unit User's Manual.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA, 1986.

[21] Ulrich Drepper, *ELF Handling for Thread-Local Storage*, Version 0.20. Red Hat Inc., December 21 2005.