# Coyotos Microkernel Specification

Version 0.2+

Jonathan S. Shapiro, Ph.D., Eric Northup, M. Scott Doerrie, Swaroop Sridhar
*Systems Research Laboratory*
Dept. of Computer Science
Johns Hopkins University

Neal H. Walfield, Marcus Brinkmann
*GNU Hurd Project*

March 20, 2006

# 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, and Dominique Quatravaux.

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

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 [4]. EROS [1] 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 [2]. All of these features were inherited with some revision from the KeyKOS system. [5] 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. The PATT concept survives in the current specification. Our main goals were cleanup, consistency, and formalization.

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 againts 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 information about the L4.sec design emerged [21], it became clear that copyable protected references might be problematic on the L4.sec interface as well. Walfield and Brinkmann travelled to Baltimore for a month-long set of design discussions in January 2006, leading to the current design for Coyotos.

The January discussions exposed a fundamental problem in the originally intended Coyotos interface: **kernel threads are not cheap.** Over the last 15 years, it has become a tenet of faith among microkernel designers that blocking and

unbuffered communication mechanisms are good. Perhaps the most persuasive argument supporting this position was offered by Liedtke in *Improving IPC by Kernel Design* [12]. The essential argument of this paper is that buffering carries additional copying costs, and that non-synchronizing IPC carries additional context switch overheads, the paper argues that the correct way to eliminate these expenses is by eliminating both features in favor of a blocking and unbuffered communication model. When an application must block on multiple communication channels, it should use multiple kernel threads. Our own research results [13] along with those of Ford *et al.* [14] and others seemed to confirm the performance argument, and unbuffered blocking IPC systems became more or less the accepted design wisdom in the microkernel world. Looking back, it now seems likely that none of us adeqautely considered the issue of space or thought hard enough about application-level complexity.

The space concern is a blatant failing in blocking designs. Suppose there is a server that wishes to wait simultaneously for activity on one of 1024 network connections. In a blocking IPC design you need a thread to wait on each of these connections. The register state of a modern Pentium-family processor occupies nearly four kilobytes, and there is additional state needed within the operating system. Estimate it at two pages, or 8 kilobytes. Ignoring their address space, runtime storage requirements, or any other space that may be needed, the mere existence of these 1024 threads requires about 8 *megabytes*. It could be argued that this is acceptable on a modern desktop PC (though I disagree). It is simply *not* acceptable on smaller embedded systems. The requirement for many threads solely for the purpose of demultiplexing presents an intrinsic failure of scalability.

The complexity problem is more difficult to see, because it occurs in relatively few programs. It is reasonably well accepted in the computer science field that concurrency is difficult for programmers to manage. When the microkernel architecture forces a server application into a multithreaded design, the server must suddenly deal with concurrency. The usual approach is to "forward" all of the requests to a single worker thread, reducing the concurrent design to a simpler, event-driven design. But now we must look more carefully at the cost of the IPC, because we have just said that a server is likely to need *two* IPCs for most incoming requests. On modern processors, the kernel/user hardware crossing delay is becoming increasingly expensive. A requirement for two IPCs simply is not a sustainable design from a performance perspective. The problem does not arise in many programs, and it is tempting to imagine that it does not need to be a focal issue for the microkernel designer. The difficulty is that the efficiency in both space and time of these programs is absolutely critical in real systems.

An alternative to the blocking approach is suggested by the sequence of work on scheduler activations in the Topaz system [15], first-class user threads in the Psyche system [16], and the Nemesis kernel [17]. The difficulty with these efforts was that nobody had worked out how to integrate them with a conventional IPC design, which is an essential underpinning of any object-based system. We believe (hope?) that the Coyotos design presented here does so.

## Coyotos: Take II

The version of Coyotos described here is an outgrowth of the January 2006 discussions between Walfield, Brinkmann, and Shapiro. In contrast to the original plan, it is an unbuffered *asynchronous* design. A single kernel-scheduled thread may have multiple receives outstanding at the same time, and will be reliably notified when each is completed.[1] Completion notifications are delivered to an application-supplied activation handler, which can elect whether to preempt the current user-mode thread of execution, enqueue the event for later processing, or simply ignore it. The resulting design presents a curious sort of hybrid: the kernel mechanisms are both asynchronous and atomic, but applications are able to efficiently construct blocking semantics on top of these if desired. If we have gotten this right, the common-case simplicity and performance of the old design has been largely preserved, but the less common (though still important) cases of multithreading and polling can now be handled sensibly.

Coyotos retains the atomicity and pure capability-based design of the EROS system, but has moved completely to this new, asynchronous communications model. It also introduces a more efficient memory mapping mechanism, jointly invented by Shapiro, Eric Northup, and Scott Doerrie. We will need to see whether the result works out. There is no substitute for measurement.

[1] Provided that it correctly implements the activation protocol.

## A Brief Word on Terminology

Coyotos is based on **scheduler activations**, a mechanism whose effect is to make user-level threads "first class." Scheduler activations permit a user-level thread scheduler to operate in cooperation with the kernel-level scheduler. This creates a significant confusion when speaking about process dispatch because two levels of dispatch now must occur: the kernel must dispatch the process, and the process's activation handler must dispatch some user-level thread. When a process re-enters the kernel, the kernel must be aware of the current process-level execution mode in order to save the process registers in a way that can later be used by the user-level activation handler. We can no longer speak simply about a process that is running or ready or idle. We must simultaneously speak about whether it is "activated" or normal.

There is also some confusion about the term **event**. It is common practice in some kernel literature to refer to events in the sense of "kernel events of interest" — things that an application may block for. In a scheduler activation design, the application instead enqueues an "event wait" structure to wait for operation completion and continues execution. There is no process state corresponding to the traditional "blocked" state. When the operation of interest completes, the enqueued event wait structures are delivered back to the interested applications in the form of an **activation**. Activations are preemptive, and cause the receiving application to transition into its activation handler in order to process the event and optionally make a user-level thread switch.

Scheduler activation designs also use events for something that we normally don't *think* of as un-blocking: the scheduling mechanism. In a conventional system, a *ready* process is actually a *blocked* process that is waiting for the CPU resource; this is just like any other type of resource blocking. In scheduler activation designs, this insight is made explicit: the transition from *ready* to *running* is signalled by an event.

In Coyotos, the event wait structure is the FCRB, and we have introduced yet another new type of event: message arrival. We have taken advantage of the first-class nature of FCRB objects to let them either be placed on "receive queues" or directly named by capabilities, and we have added state to the FCRB that allows it to name a "receive block" in the recipient process describing where an incoming message payload should be placed. This allows us to break IPC into two phases: payload transfer, which is atomic but happens "in the background" from the perspective of the receiving process, and the message arrival event, which is signalled as an activation.

This is a new idea, and we are still sorting out the correct terminology in this specification. If you are confused, it may very well be sloppy terminology.

Jonathan S. Shapiro, Ph.D.
Department of Computer Science
Johns Hopkins University
January, 2006

# 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, PATTs (mapping structures), schedules, first-class receive buffers (FCRBs), receive queues, wrappers, pages, and a small number of other kernel objects.

**Processes** Processes are the unit of execution, scheduling, and resource binding. A process names two address spaces: one for data (bytes) and the other for authorities (capabilities). Processes also name their 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.

**PATTs** PATTs 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 PATTs whose leaf capability slots name atomic storage units. A virtual address is divided into a **virtual page address** and a **page offset**. In the case of data address spaces, valid virtual page addresses describe paths to leaf slots that contain data page capabilities. In the case of capability address spaces, valid virtual page addresses describe paths to leaf slots that contain capabilities to capability pages.

**First-class receive buffers (FCRBs)** FCRBs capture all of the information necessary for a process to receive an incoming event of interest. FCRBs are similar to the event wait structures of Nemesis [17], but have been extended to provide a mechanism for message transfer as well. The messaging aspect bears some resemblance to the first-class messages of HYDRA [18]. In certain cases, FCRBs can be used to provide a guaranteed non-blocking, single-delivery notification to the receiving process. This resolves several of the control flow denial of service difficulties that are present in EROS and other systems based on unbuffered, blocking communication systems [3].

**Receive Queues** Receive queues allow multiple FCRBs to be enqueued simultaneously to wait for some incoming message. When a message is sent via the receive queue, the kernel will select a waiting FCRB from the queue and deliver the message to the receiver named by the FCRB. This permits kernel demultiplexing of receive processes, which enhances performance on multiprocessos. Receive Queues can also be used in certain circumstances as a group synchronization primitive.

**Wrappers** Wrappers permit an arbitrary capability to be "wrapped" in such a way that it can be individually revoked. Wrappers provide a kernel implementation of Redell's caretaker mechanism [19]

**Pages** Pages are the atomic unit of data and capability storage allocation. An address space consists of a lattice of PATTs 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 Sender Capabilities

Several object capabilities, notably FCRBs, Receive Queues, and Wrappers, have "sender" capabilities. These capabilities do not implement operations on the corresponding kernel abstractions. Instead, they provide the means by which an application introduces new services. Any invocation of a sender capability is encapsulated as a message that is delivered to a server.

## 1.3 Activations

The Coyotos design is based on *scheduler activations*. From the kernel perspective, a process has three execution states: **idle**, **ready**, and **running**. The traditional "blocking" state is subsumed by an asynchronous messaging mechanism. The message delivery that is enqueued until the completion of a kernel operation rather than the requesting process.

At user level, a process has two states in which it may be executing: the **normal** state and the **activated** state. Incoming events are always delivered in association with a transition from the normal to the activated state. The activated state has a separate entry point known as the *activation handler*. The activation handler exists to make dispatching (or deferral) decisions about incoming events. One possibility is that the activation handler may elect to perform a user-level thread switch, and therefore acts as a user-mode scheduling agent (which is how these came to be known as *scheduler activations*. The user-level thread switch is kernel supported by placing the register save area for user-mode registers in memory that is shared between the kernel and the application and is readable and writable to both. The kernel supports the activation handler by handling register save in a mode-sensitive way on kernel entry. A scheduler activation may be thought of as a kernel-implemented trap mechanism that operates entirely at application level. It is similar in concept to a single-level hardware exception mechanism in a RISC processor.

Another possible handling mechanism is that the activation handler might simply enqueue the newly received message for later processing and then resume the current user thread of control without modification. In the latter design, the user thread would typically execute in event-driven fashion until all pending events are processed.

Notifications of all kernel events of interest, including kernel scheduling events, exceptions, and arriving messages, are delivered to the process using activations.

## 1.4 Messages

Coyotos provides two types of messages: idempotent and stateful. 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.

**Idempotent Messages** An idempotent message is one whose content is set by the receiver (in the FCRB). The information conveyed by the arrival of an idempotent message is the fact that it has been transmitted (fired) one or more times by one or more senders since it was last received. If the receiver is currently executing an activation from a previous transmission of the same idempotent message at the time of a subsequent send, the idempotent message will be delivered again. Sends on idempotent message FCRBs do not block.

**Stateful messages** A stateful message is one where the sender defines the message payload and the receiver specifies (via a stateful FCRB) the location(s) in the receiver address space where the payload should be delivered. A stateful receive FCRB may be either *pending* or *delivered*. A send operation on a stateful FCRB will not make progress until that FCRB becomes pending. On completion, it causes the FCRB state to transition from pending to delivered. The receiver must explicitly reset the FCRB state before subsequent attempts to send on that FCRB will make progress. This ensures that the received message payload cannot be overwritten by successive senders before processing.

**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. Senders may implement watchdog timeouts on send operations by arranging to send themselves an idempotent message after a preset 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.

## 1.5 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 an optional 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 dropped. The asynchronous receive phase, if present, causes a first-class receive buffer to be unblocked for later asynchronous delivery. Finally, the invoker may optionally elect to cease processing instructions. When combined with an asynchronous receive, this has the effect of blocking until the next incoming event.

The Coyotos kernel implements only one system call: `InvokeCapability`. Additional kernel-implemented instructions may be introduced by particular implementations. The distinguishing characteristic between an invocation and a kernel-implemented extended instruction is that the extended instruction is generally a fast-path construct, and can induce a context switch only if an exceptional condition arises during the operation. A second distinguishing characteristic is that extended instructions do not explicitly invoke a capability, but instead make use of the protection domain defined by the process executing the extended instruction.

Sender capabilities to FCRBs, receive queues, or wrappers contain a 32-bit protected payload field. The protected payload field is delivered to the recipient as part of event delivery, and is neither readable nor modifiable by the capability's invoker (which is why it is said to be protected). Servers may use the protected payload to distinguish interfaces, object identities, or any other desired characteristic.

## 1.6 Exception and Interrupt Handling

The Coyotos kernel does not specify or implement a policy for 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 as an activation, either to the offending process's activation handler or to an external fault handler designated by an FCRB.

Coyotos distinguishes two categories of exceptions: addressing exceptions and all others. Exception handlers can be configured that receive and optionally handle the addressing exceptions arising from references within any naturally aligned $2^k$ page subregion of an address space. This allows application level code to implement copy on write, demand loading, demand zero, and similar functions that have traditionally been implemented directly by monolithic kernels.

## 1.7 Protection Model

**Pending Edit**

*There is a missing "theory of operation" discussion here about what the word "trust" means. This needs to be dealt with and the text re-written accordingly.*

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. 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 [20].

As an experimental tool for supporting application-level implementation of labeled policies, Coyotos also implements per-process labels. These labels are a form of process-associated "protected payload" that is transferred to the receiver as part of the capability invocation protocol. An invoking process may elect not to disclose its protected payload, but a receiving process can detect when this has been done. The sending process is unable to *alter* its protected payload. This allows the operating system to define per-process labels that can be relied on by servers implementing the services of that operating system.

# 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 background mapping window (see the Address Spaces chapter of Microkernel Abstractions). | RO,NX,WK |
| 2 | KeyBits | Discloses the bit representation of capabilities. | |
| 3 | Discrim | Classifies capabilities. | |
| 4 | Range | Fabricates object capabilities. | |
| 5 | Sleep | Interface to the kernel interval timer. | |
| 6 | IRQ Control | Interrupt request line control interface. | |
| 7 | Schedule Control | Interface to the kernel master scheduling table. | |
| 8 | Checkpoint | Control capability for the kernel checkpoint mechanism. | |
| 9 | ObStore | Interface between kernel and object store manager. | |
| 10 | I/O Permission | Permission to perform I/O instructions. | |
| 11 | Pin Control | Permission to pin objects in memory. | |
| *12-33* | *Reserved* | *Encodings reserved for future use.* | |
| 32 | FCRB | Control capability for a first-class receive buffer. | |
| 33 | Receive Queue | Control capability for a receive queue. | |
| 34 | Wrapper | Wraps any existing capability. | |
| 35 | Page | Data page. The size of a page is determined by the underlying hardware page size. | RO,NX,WK |
| 36 | Capage | Capability page. The size of a capability page is determined by the page size of the underlying hardware page size. | RO,NX,WK |
| 37 | PATT | Prefixed address translation table. Used to compose larger address spaces from pages. | RO,NX,WK |
| 38 | Opaque PATT | See discussion of PATTs. | RO,NX,WK |
| 39 | Process | Capability that manipulates the kernel process abstraction. | |
| *40-60* | *Reserved* | *Encodings reserved for future use.* | |
| 61 | FCRB Sender | Authority to send to the process designated by an FCRB. | |
| 62 | Receive Queue Sender | Authority to send to some FCRB enqueued on the designated receive queue. | |
| 63 | Wrapper Sender | Authority to send a message via some wrapped capability. | |

The **RO**, **NX**, and **WK** restrictions respectively indicate, read-only, non-executable, and weak 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. Certain capabilities have a "hazard" bit that is encoded using the least significant bit of their type field as noted above. For these capabilities, the odd type code will never appear outside the kernel.

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, capage, PATT, and window capabilities. All of these are used to describe portions of the address space. The format of page, capage, and PATT capabilities is:

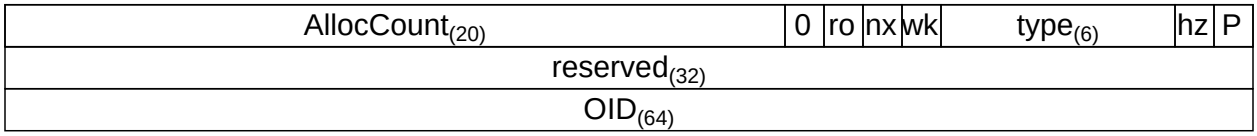


Figure 2.1: Memory object capability

The format of a window capability is:

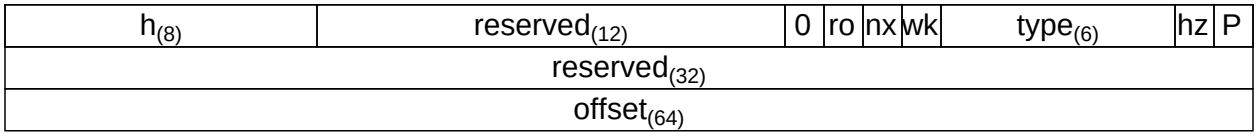


Figure 2.2: Mapping window capability

The h field defines the span of the mapped window. Its use and meaning are described in the Address Translation chapter. A well-formed window capability's offset value will always be a multiple of $2^h$.

### 2.1.2 Message-Related Capabilities

FCRB, Receive Queue, and wrapper 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.

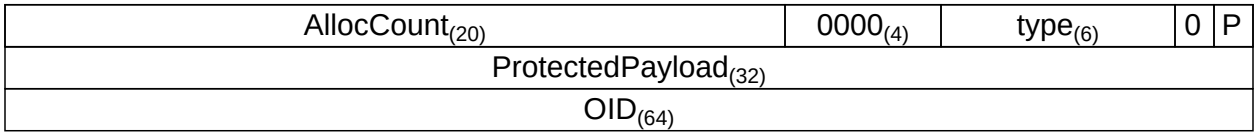


Figure 2.3: FCRB, receive queue, or wrapper capability

Invocations of the "normal" forms of these capabilities ignore the protected payload and provide access to the kernel-implemented object. Invocations of the "sender" versions of these capabilities make use of the protected payload field.

### 2.1.3 Capabilities to Processes

The format of a process capability is:

| AllocCount$_{(20)}$ | 0000$_{(4)}$ | type$_{(6)}$ | 0 | P |
|---|---|---|---|---|
| reserved$_{(32)}$ | | | | |
| OID$_{(64)}$ | | | | |

Figure 2.4: Process capability

### 2.1.4 Miscellaneous Capabilities

The format of a miscellaneous capability is:

| reserved$_{(20)}$ | 0000$_{(4)}$ | type$_{(6)}$ | 0 | P |
|---|---|---|---|---|
| reserved$_{(96)}$ | | | | |

Figure 2.5: Miscellaneous capability

## 2.2 Validity

Those capabilities whose representation includes an allocation count name resources that are revocable. When the target resource is revoked, all capabilities naming that resource become Null capabilities.

Capabilities are commonly stored on disk. Efficient revocation of capabilities when a resource is destroyed relies on the allocation count. In order for the capability to be valid, the allocation count of the capability must match the allocation count of the object at the time of the invocation. If the allocation counts do not match, attempts to invoke the capability behave as if the Null capability had been invoked.

The null capability is a capability that responds only to the getType() operation.

## 2.3 Extensibility

Coyotos is an extensible object system in the sense of Hydra [18]. New objects may be introduced by designing a process that implements the desired object. Capabilities to these objects are implemented as Sender's capabilities. The kernel checks these capabilities for validity, and optionally for a protected payload match (see Messages and FCRBs), but does not otherwise define semantics for these capabilities.

Because the kernel does not know the semantics of these extensions, sender capabilities are not considered "safe" by the `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 for execution at the kernel level.

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 data and capabilty address spaces. This achieves concurrent execution of multiple kernel threads of control within a common addresssing environment and resource pool.

All Coyotos system calls return asynchronously using an event wait structure known as an FCRB. The notion of a process blocked for completion of a kernel operation is subsumed by enqueuing an FCRB on the associated kernel operation completion queue while the requesting process continues execution. When the kernel operation completes, an event notification is delivered via the enqueued FCRBs.

## 3.1 State of a Process

The state of a process includes both "user state" and "supervisor state." Kernel state is stored in the process itself. User state is stored in the user process control block **UPCB**.[1]

| |
|---|
| protPayload$_{(32)}$ |
| faultCode$_{(32)}$ |
| faultInfo$_{(32/64)}$ |
| schedule$_{(cap_t)}$ |
| dataSpace$_{(cap_t)}$ |
| capSpace$_{(cap_t)}$ |
| ioSpace$_{(cap_t)}$ |
| brand$_{(cap_t)}$ |
| handler$_{(cap_t)}$ |
| upcb$_{(cap_t)}$ |
| capRegs$_{(cap_t)}$ |

Figure 3.1: Privileged Process State

The process structure holds state that must be be protected by the kernel. Per-process capability state, fault code, and

[1] I have reluctant decided *not* to adopt the L4 terms KTCB and UTCB because the abbreviation "TCB" for "thread control block" induces tremendous confusion in discussions of security architectures.

fault information can be accessed and manipulated only through invocations of kernel-implemented capabilities. The meanings of the process fields are:

**protPayload** The per-process protected payload (experimental).

**schedule** Capability naming the process's scheduling contract.

**dataSpace, capSpace, ioSpace** Capabilities naming (respectively) the process's data address space, capability address space, and I/O address space.

**brand** Capability that identifies this process to its creators. This capability can be overwritten through the process capability, but cannot be read.

**handler** FCRB capability to the per-process fault handler.

**upcb** Capability to a data page containing the upcb data.

**capRegs** Capability to a capability page containing the per-process capability registers.

The UPCB is typically mapped in read-write form at a process-determined location within the process's data address space. This data structure is processor-dependent, but has some elements that are common across all architectures. All architectures are required to define hardware registers corresponding to the `R0..R3` slots, but the particular hardware registers defined are architecture-specific, and the layout and size of the `other_hw_registers` region is entirely architecture-specific.

| other_hw_registers$_{(???)}$ | | |
|---|---|---|
| dispatchR3$_{(32/64)}$ | | |
| dispatchR2$_{(32/64)}$ | | |
| dispatchR1$_{(32/64)}$ | | |
| dispatchR0$_{(32/64)}$ | | |
| dispatchPC$_{(32/64)}$ | | |
| dispatchSP$_{(32/64)}$ | | |
| userR3$_{(32/64)}$ | | |
| userR2$_{(32/64)}$ | | |
| userR1$_{(32/64)}$ | | |
| userR0$_{(32/64)}$ | | |
| userPC$_{(32/64)}$ | | |
| userSP$_{(32/64)}$ | | |
| actSP$_{(32/64)}$ | | |
| actPC$_{(32/64)}$ | | |
| reserved$_{(16)}$ | PendingEvents$_{(8)}$ | Activated$_{(8)}$ |

Figure 3.2: UPCB State

The meanings of the UPCB fields are:

**other_hw_registers** Saved locations for architected registers not otherwise explicitly saved. Content is architecture dependent.

**dispatchR0...dispatchR3** R0..R3 values that will be reloaded when process is next dispatched.

**dispatchPC, dispatchSP** PC and SP that will be reloaded when the process is next dispatched.

**userR0...userR3** Save locations for R0..R3 when entering kernel from normal execution.

**userPC, userSP** Save location for PC and SP when entering kernel from normal execution.

**actPC, actSP** Entry point and stack pointer for the activation handler.

**PendingEvents** Set to 1 by the kernel whenever a new incoming event is enqueued. Cleared to 0 by the kernel when an event is successfully delivered and there are no further pending events.

**Activated** Set to 1 by the kernel when beginning activated execution. Set to 0 by the application when transitioning from activated to normal state.

## 3.2 FCRBs

The FCRB structure encapsulates waiting for events. An FCRB is associated with a receiving process by the process capability stored in the `recipient` slot. When a kernel operation completes, all FCRBs that have been placed on the associated completion queue are released to their associated processes. Alternatively, an FCRB sender's capability can be invoked directly with the same effect.

FCRBs come in three forms: idempotent, short, and long. An idempotent FCRB (ID=1) has no writable state. The values of the `w0`, `w1`, `extRcvDescrip`, and `protectedPayload` are established by the receiver at the time of FCRB creation, and are not altered at the time of the event. An idempotent FCRB can be sent multiple times without blocking, but may be received fewer times than it is sent. It is guaranteed that an idempotent event will be delivered at least once after a send operation, provided the receiving process continues to exist.

Long and short FCRBs are stateful (ID=0). A short FCRB (MS=1) provides two words of payload: `w0` and `w1`. A long FCRB (MS=0) provides for a longer payload. Stateful FCRBs may be blocked (BL=1). Attempts to invoke a stateful FCRB block until the FCRB becomes unblocked (BL=0). When a stateful FCRB is used to send a message, the FCRB becomes blocked (BL:=1). This ensures that the arriving message state can be processed before it is overwritten by subsequent sends.

An FCRB carries the following state:

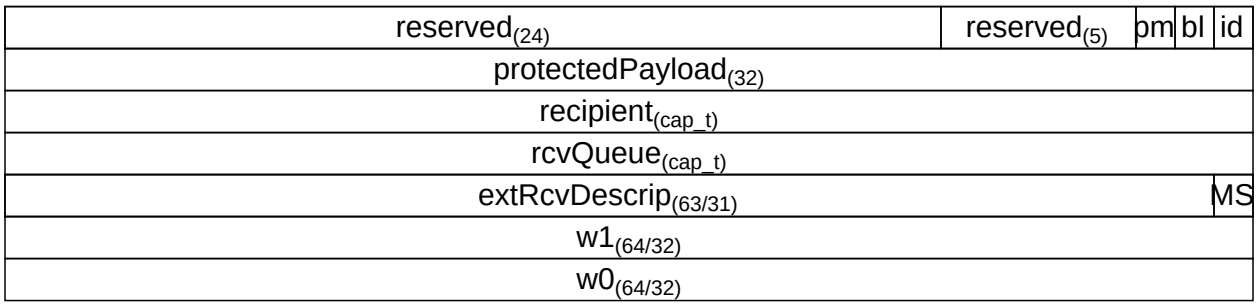


Figure 3.3: FCRB State

The meanings of the FCRB fields are:

**pm Payload Match** Determines whether a payload matching check will be performed.

**bl Blocked** Describes whether the FCRB is currently blocked.

**id Idempotent** Determines whether user-supplied message payload is accepted or ignored for this FCRB. Multiple sends of an idempotent FCRB may be collapsed into one.

**recipient** Process capability to receiving process.

**rcvQueue** Control capability to receive queue that this FCRB should be enqueued on when not blocked.

**protectedPayload** Used for matching, and/or to temporarily store the protected payload from the sender capability.

**MS Message Size** Used to determine whether the **extRcvDescrip** pointer is valid.

**extRcvDescrip** Pointer to the extended receive descriptor if the message size field is zero. User-defined if the message size bit is 1. This field is discussed in the chapter on Capability Invocation.

**w0, w1** Transient storage for first two transmitted register arguments of message.

## 3.3 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 `InvokeCapability` instruction.

From the kernel perspective, a process exists in one of three execution states: **idle**, **ready**, or **running**. The transition diagram is:

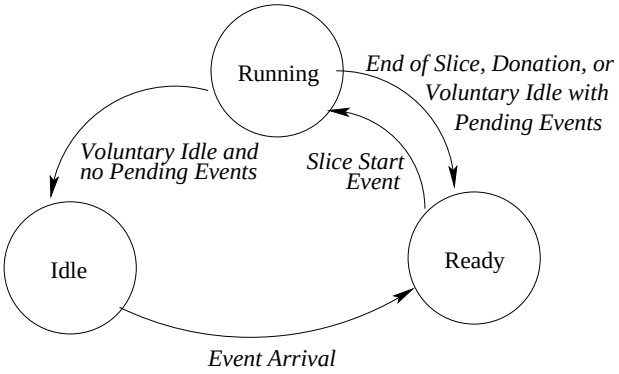


Figure 3.4: Process state transitions

From the user perspective, a process may either be in the **activated** or **normal** states according to the current value of the `Activated` field of the UPCB. These states are discussed below.

### 3.3.1 Event Arrival and Run-In

Event arrival processing is always handled on the local processor, so the receiving process is necessarily **ready** or **idle** at the time of event arrival. When an event arrives for a process, the **PendingEvents** field is atomically set to 1, and the FCRB associated with the event is enqueued on the process for delivery. If the process is in the **idle** state, it moves to the **ready** state.

The transition from the **ready** state to the **running** state is prompted by the arrival of a **run-in** event, which is an idempotent message type. A consequence of this design is that there is *always* at least one event enqueued for delivery when the process transits from **ready** to **running**. In order to transition from the **ready** state to the **running** state, processing proceeds as follows:

1. The **Activated** field is tested.

   - If the **Activated** field is zero, the process is currently in the normal state. An arbitrary enqueued event is chosen for delivery (as described below), and its payload is transferred to the *dispatch* registers.
     The **Activated** field is set to 1.

2. If further events are queued for delivery, the **PendingEvents** field is set to 1. Otherwise, it is set to zero.

3. The process is dispatched to a CPU and execution begins.

Implementations are encouraged to integrate the event arrival and run-in transitions in a single path where possible. It is a very common case that the receiving application is idle or running in non-activated state at the time of event arrival.

### 3.3.2 Preemption

When a process goes from the **running** state to either the **ready** or **idle** states, the current setting of the **Activated** field determines how the save area will be constructed:

- If the **Activated** field is 0, then the registers **R0..R3, PC, SP** are saved to the corresponding *user* fields, and the **dispatchPC** and **dispatchSP** values are copied from **actPC** and **actSP**. This will lead to a transition from normal to activated execution on resumption.
- If the **Activated** field is non-zero, then the registers **R0..R3, PC, SP** are saved to the corresponding *dispatch* register fields and the *user* fields are unchanged. This corresponds to resumption of a preempted activation handler.

### 3.3.3 Exception Handling

There are two cases in which a process may need to process an exception: execution exceptions and message transfer exceptions. An execution exception occurs synchronously as the result of an instruction executed performed by the process. A message transfer exception occurs asynchronously as a side-effect of the payload transfer associated with an incoming message.

In either case, the handling of exceptions depends on the current setting of the **Activated** field.

- If the **Activated** field is 0, then the exception has occurred while the application is running in “normal” mode. In this case, the fault code and auxiliary fault information associated with the exception will be delivered to the activation handler as if by an event arrival.
- If the **Activated** field is non-zero, then the exception has occurred in the activation handler itself. In this case the kernel will synthesize a message to be delivered as if by invoking the **handler** capability stored in the process structure. The fault code and auxiliary fault information will be passed as arguments to this invocation.

## 3.4 Event Delivery

While the particular registers selected to convey information to the activation handler are architecture-specific, the general model of activation handling is common to all architectures.

An event will be delivered only as the receiving process is entering the **activated** state. Every event has an associated FCRB. Delivery proceeds by copying the **extRcvDescrip**, **protectedPayload**, **w0**, and **w1** fields to (respectively) the **dispatchR0**, **dispatchR1**, **dispatchR2** and **dispatchR3** registers of the UPCB before dispatching the receiving process.

# Chapter 4

# Address Spaces

A Coyotos process has three address spaces: a data address space, a capability address space, and an I/O address space.[1] The capability and data address spaces are byte addressable. Capabilities in the capability address space may only be stored or referenced from naturally aligned addresses. Much of the discussion that follows is identical for data and capability address spaces. Where this is the case we use parenthesis to indicate the capability variant.

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, capages, and PATTs. 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:

- In addition to address translation, the Coyotos mapping structures provide support for context-sensitive fault handlers. The per-process fault handler may optionally forward memory fault messages to these memory fault handlers in order to request object-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.

[1] This may increase to three if we introduce I/O address spaces.

### 4.1.1 Permissions

All memory object capabilities carry three bits that specify restrictions on what type of access may legally be performed:

**NX No Execute** Instruction fetch references along any address translation path that traverses this capability are prohibited. Attempts to perform instruction fetches at such addresses generate an `ExAccessViolation` 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.

**RO Read Only** Attempts to perform write references along any address translation path that traverses this capability are prohibited. Attempts to perform instruction fetches at such addresses generate an `AccessViolation` exception.

**WK Weak** 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.

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 reported, the type of the exception and the originally referenced address are provided to the referencing process's activation handler, the faulting instruction (if any) has no effect), and the program counter (if the access is synchronous) is not advanced. The defined reference types are:

| Type | Meaning |
|---|---|
| SDR | **Synchronous Data Read** Generated by a data load instruction. |
| ADR | **Asynchronous Data Read** Generated by a read operation on an extended receive descriptor during asynchronous message transfer. |
| SDW | **Synchronous Data Write** Generated by a data store instruction. |
| ADW | **Asynchronous Data Write** Generated by an arriving message transfer. |
| IF | **Instruction Fetch** Generated by a hardware instruction fetch. All instruction references are synchronous. |
| SCR | **Synchronous Capability Read** Generated by a capability load instruction or the send phase of an invocation. |
| SCW | **Synchronous Capability Write** Generated by a capability store instruction. |
| ACW | **Asynchronous Capability Write** Generated by an arriving message transfer. |

### 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.

The resulting effective restrictions may lead to an exception according to the following rules:[2]

[2] *Documentation issue:* Marcus wonders if there is not a simpler way to present this than a table.

| Ref Type | Cap Type | Cap Perms | Result |
|---|---|---|---|
| SDW,ADW | Page | RO | Exception: `AccessViolation` |
| SDW,ADW | Capage | *any* | Exception: `AccessViolation` |
| IF | Page | NX | Exception: `ExAccessViolation` |
| IF | Capage | *any* | Exception: `ExAccessViolation` |
| SCW,ACW | Capage | RO | Exception: `AccessViolation` |
| SCR,ACR | Capage | wk | Access permitted, result is weakened. |
| SCW,ACW | Page | *any* | Exception: `AccessViolation` |
| SDR,ADR | Capage | *any* | Exception: AccessViolation. |
| SDR,ADR | Page | *any* | Access permitted. |
| SCR,ACR | Page | *any* | Exception: AccessViolation. |
| SCR,ACR | Capage | *any* | Access permitted. |

All discussion of exceptions generated by translation in the following sections are *in addition* to the exceptions associated with access violations. In order for a reference to proceed without generating an exception it must be both permitted and valid.

## 4.2 Pages

The smallest mappable unit, and therefore the smallest address space, is the page (capage). Coyotos implements a single page size whose size matches some hardware page size implemented by the underlying hardware. 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 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.

A process that can execute entirely within a single page[3] may place a page (capage) capability into its data (capability) address space slot, with the effect of defining an address space having valid offsets between [0,*pgsize*-1]. Attempts to reference offsets beyond this page 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. Except where explicitly documented as having kernel-defined semantics, the least significant 4 bits of a capability address are not considered significant by the kernel.[4]

Address translation of an address *addr* with respect to a page or capage 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 PATT object. A PATT consists of a height (**h**), a residual (**r**), a vector of non-overlapping **patterns**, and a vector of capabilities. The state of a PATT is shown below. The pattern and capability vectors are positionally correlated: the pattern stored at `patvec[2]` is semantically linked to the capability stored at `capvec[2]`.

**Invariants** The pattern values *pat* stored in `patvec` must be unsigned quantities in the range $0 \leq pat < 2^{h}$. This restriction is enforced by the pattern manipulating operations on the PATT capability.

[3] In practice this case is hypothetical, because it would require the process to execute out of the available storage at the top of the UPCB page. This is possible on some architectures, but it is probably more trouble than it is worth.

[4] I do not like this rule. I would strongly prefer requiring an alignment fault for misaligned capability addresses, but the cost of implementing this in the kernel path seems likely to be substantial.

| | | |
|---|---|---|
| $OID_{(64)}$ | | |
| $allocationCount_{(24)}$ | | $reserved_{(8)}$ |
| $reserved_{(16)}$ | $h_{(8)}$ | $r_{(8)}$ |
| $patvec[15]_{(64/32)}$ | | |
| ... | | |
| $patvec[0]_{(64/32)}$ | | |
| $capvec[15]_{(cap\_t)}$ | | |
| ... | | |
| $capvec[0]_{(cap\_t)}$ | | |

Figure 4.1: PATT State

### 4.3.1 PATT Translation

Address translation of an address *addr* with respect to a PATT is defined as follows:

1. A temporary address *addr'* is computed:

```
addr' := addr & ~((1u << r)-1u);
```

2. This temporary address is compared to all of the stored `patvec` values in parallel.

   - If no patvec value matches, the address is invalid and an `InvalidAddress` exception is generated.
   - Otherwise, there exists some *i* such that `patvec[i]` matched. The result of the address translation is the result of translating `addr % (1u<<r)` with respect to the address space defined by the capability stored in `capvec[i]`.

A hidden consequence of this definition is that attempts to translate an address *addr*$\geq 2^h$ yields an `InvalidAddress` exception. That is, the value $h=\log_2(span)$ where *span* is the maximum number of bytes that are addressable in the subspace dominated by this PATT.

### 4.3.2 Fault Handler and Background Window Context

Any `patvec` values having non-zero bits in the least **r** positions are unmatchable by the matching procedure. Certain pattern values have reserved meanings:

| Value | Use | Description |
|---|---|---|
| 1 | Background Space | Window capabilities appearing in the current PATT or below will be interpreted as offsets with respect to the address space capability stored in the corresponding `capvec` slot. |
| 2 | Handler | Corresponding `capvec` slot names a fault handler FCRB. An address fault message generated by the `probe()` method will be delivered to the nearest enclosing fault handler defined in this way. |
| 16 | Free Entry | This slot of the PATT is not in use. |

The rule for processing address probe faults and window capabilities is to use the "nearest enclosing context". A fault message is delivered by logically traversing the address space mapping structures until a page or an undefined subspace is discovered, and then delivering the fault message to the nearest enclosing keeper. Similarly, the interpretation of a background window capability is performed with respect to the nearest enclosing background space.

### 4.3.3 Window Translation

If a address translation traverses a `capvec` slot containing a window capability, translation of the address *addr* proceeds as follows:

1. If *addr*$\geq 2^{\texttt{h}}$, where `h` is taken from the window capability, then no valid translation exists and an `InvalidAddress` exception is generated.
2. Otherwise: *addr* is combined with the *offset* field of the window capability. The result of the address translation is the result of translating `(offset|addr)` with respect to the nearest enclosing background space capability.

Recall that a well-formed window capability offset value will always be a multiple of $2^{\texttt{h}}$.

### 4.3.4 Cycle Detection

It is possible for an erroneous or hostile program to arrange PATT objects and their `capvec` capabilities 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 PATT 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 PATT. 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 PATT structures *that do not translate new bits* that will be visited before generating a `MalformedSpace` exception.

   This method keeps track of $\texttt{r}_{\texttt{least}}$, the least **r** value that has been traversed to date. If $\texttt{r}_{\texttt{patt}} \geq \texttt{r}_{\texttt{least}}$, then the current PATT 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 PATT. 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 `(h-r)` for all PATTs 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 implementors: 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 a sender 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 sender 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 PATT for the subspace. For example, in a system having 4 kilobyte pages, the sender may wish to transmit a $2^{11}$ page subspace, but the subspace may currently be dominated by a PATT having `h=25` and `r=21`. Before a single dominating capability can be sent, this PATT must be "split" into an arrangement where the target subtree has a dominating PATT with `h=23`. When such a send is attempted, the sender will receive a `SplitFault` exception. This is an advisory that the PATT must be split in order to bring a dominating PATT into existence.

Similarly, if a receiver specifies a "hole" of some size $2^{\texttt{hlsz}}$ pages, there must exist some PATT in the receiver tree such that:

- $\texttt{r}_{\texttt{patt}} \leq \texttt{hlsz} \leq \texttt{h}_{\texttt{patt}}$
- The receiving PATT either has some slot free or some pattern that exactly matches the target address.
- The receiver has write permissions on the appropriate PATT slot. This requires both that the permissions check described above are satisfied and that no PATT capability on the path from the address space root to the modified PATT is opaque.

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 and Messages

All operations on objects in Coyotos are initiated by the `InvokeCapability` system call. A capability invocation consists of three phases:

- The send phase, which is atomic. Sends may be performed in normal or non-blocking fashion. Blocking may occur if accessing the receiver's extended receive area requires exception resolution or if the receiving FCRB is blocked. The payload of the send may be partially or wholly dropped if complete delivery would require the sender to block and the sender is unwilling to do so.
- The receive setup phase, which is atomic and non-blocking. The Coyotos receive phase consists of identifying a receive FCRB through which a subsequently received message should be accepted.
- The completion phase, which is atomic, and allows the process to voluntarily release or donate the processor at the end of the invocation.

All three phases are optional. The invoked capability is most commonly a sender's capability to an FCRB, receive queue, or wrapper, but may also be a kernel capability.

## 5.1 The Invocation Block

The `InvokeCapability` operation takes a capbility to invoke, a message payload, and a number of control fields. The message payload consists of $4 <= n \leq 64$ virtual data registers, 4 capability items, and an optional indirect data string up to 65536 bytes in length.[1] Virtual data registers VDR0 and VDR1 are transmitted by the sender and delivered to the receiver in hardware registers.

**Consistency Issue**

The send block has an ncap field, but here we say exactly four. We need to pick a consistent position.

### 5.1.1 Control Word

Invocation is controlled by a register-based invocation control word of the form:

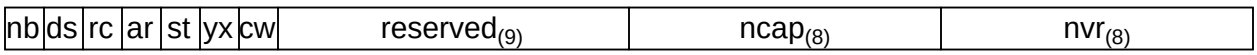


Figure 5.1: Invocation Control Word

The meaning of the bits in the invocation control word are as follows:

[1] We are considering adding an optional indirect capability string up to 65536 bytes (4096 capabilities) in length if it proves well motivated.

| Bit | Name | Meaning |
|---|---|---|
| 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. |
| DS | Data String | If set (1), an indirect string should be transmitted as part of the message. |
| AR | Activate Receive | If set (1), the `ReceiveCap` names an FCRB or endpoint control capability that should be activated following the send phase. |
| YX | Yield Execution | If set (1), sender does not wish to continue executing following the end of the current slice. If the ST field is clear (ST=0), the slice ends voluntarily at the end of the receive phase. If the ST field is set (ST=1), the receiver will execute for the remainder of the current slice. If a process having pending events goes voluntarily idle without donating its slice, the events will immediately be delivered to the activation handler. Otherwise, the highest priority eligable process known to the kernel scheduler will be dispatched. |
| ST | Slice Transfer | If set (1), donate the remainder of the sender's slice to the receiver. |
| RC | Reply Capability | If set (1), and `ncap[0]` names an FCRB or endpoint control capability, `ncap[0]` will be replaced with the corresponding sender's capability, and the protected payload field of `ncap[0]` will be overridden by the sender-provided protected payload. |
| CW | Closed Wait | If `ReceiveCap` names an FCRB, this bit is copied to the **PM** bit of the FCRB and ProtectedPayload is copied to the protected payload field of the FCRB. |
| ncap | Num Caps | Number of capability items present in the send block descriptor. **Invariant** `ncap`$\leq 4$. |
| nvr | Num Registers | Number of virtual data registers to be transmitted. Values lower than 2 are treated as 2. **Invariant** `nvr`$\leq 64$. |

### 5.1.2 Message Send Block

In order to send a message, the sender formulates a **message send block** on the stack. The components of the send block depend on which bits of the control word are set, but a fully elaborated send block takes the form:

| |
|---|
| VDR[nvr-1]$_{(64/32)}$ |
| ... |
| VDR[0]$_{(64/32)}$ |
| strPtr$_{(64/32)}$ |
| strLen$_{(32)}$ |
| capStrPtr$_{(64/32)}$ |
| capStrLen$_{(32)}$ |
| ProtectedPayload$_{(32)}$ |
| cap[3]$_{(capitem\_t)}$ |
| ... |
| cap[0]$_{(capitem\_t)}$ |

Figure 5.2: Message send block

The `Cap` vector is examined only if the value of `ncap` is non-zero in the invocation control word. The `StrPtr` and `StrLen` fields are examined only if the `SS` bit of the invocation control word is non-zero. The `protected payload` field is examined only if the `RC` or `CW` bits of the invocation control word are set.

### 5.1.3 Meaning of `capitem_t`

The `capitem_t` type describes a capability location, and takes the form:

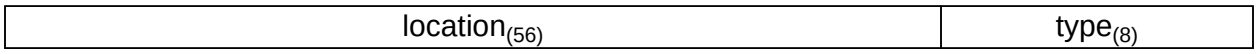


Figure 5.3: Meaning of `capitem_t`

Where the meaning of `location` depends on the value of the `type` field:

| Value | Type | Location Meaning |
|---|---|---|
| 0 | Null | Transmitted capability is a null capability. Location field is ignored. |
| 1 | Register | Capability is sent from or received to a capability register. |
| 2 | Address | Location describes an address relative to the sending/receiving process's capability address space. |
| 128..183 | Data Region | Capability to a $2^k$ region of the data address space. The `location` must be multiple of $2^{(\texttt{type-128})}$. In a recieve descriptor, this type describes a receive hole. |
| 192..246 | Capability Region | Capability to a $2^k$ region of the capability address space. The `location` must be multiple of $2^{(\texttt{type-192})}$. In a recieve descriptor, this type describes a receive hole. |

### 5.1.4 Other Parameters

In addition, there are a number of items that need to be placed in either registers or the send message block, but I haven't figured out which yet:

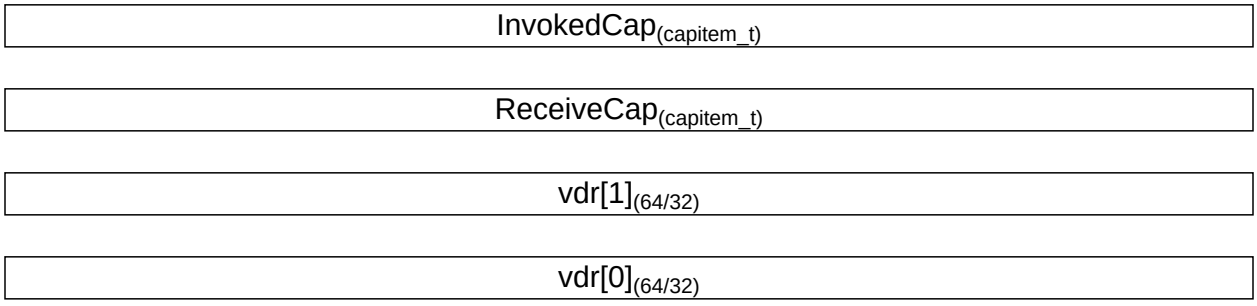


Figure 5.4: Other Parameters

## 5.2 The Receive Block

The receive block is the dual of the send block. It is referenced by a stateful FCRB, and accepts the parameters sent by the sender.

On receive, the first two virtual registers are (logically) written to the receiving FCRB and later delivered to the application in registers. The remaining parameters are written according to the message receive block.

### 5.2.1 Message Receive Block

The layout of the extended receive descriptor is similar to the organization of the message send block.

**Open Issue**

| | | |
|---|---|---|
| VDR[nvr-1]$_{(64/32)}$ | | |
| ... | | |
| VDR[0]$_{(64/32)}$ | | |
| rcvPtr$_{(64/32)}$ | | |
| rcvLimit$_{(32)}$ | | |
| rcvCapStrPtr$_{(64/32)}$ | | |
| rcvCapStrLimit$_{(32)}$ | | |
| ProtectedPayload$_{(32)}$ | | |
| cap[3]$_{(capitem\_t)}$ | | |
| ... | | |
| cap[0]$_{(capitem\_t)}$ | | |
| rcvLen$_{(32)}$ | | |
| sndLen$_{(32)}$ | | |
| rcvCapLen$_{(32)}$ | | |
| sndCapLen$_{(32)}$ | | |
| reserved$_{(16)}$ | rcap$_{(8)}$ | rvr$_{(8)}$ |

Figure 5.5: Message receive block

Where should the application place information that supports the activation handler in associating an incoming message with a user-level thread?

The `VDR[0]`, `VDR[1]`, and `ProtectedPayload` values are returned to the process via hardware registers during activation. Slots are preserved for them in the message receive block to facilitate context save if message processing is to be deferred. The `RcvLen` and `SndLen` fields are updated only if a string is being accepted. The `Cap` fields are consulted only if capabilities are being accepted. Positions in the `VDR` vector exceeding the value of `nvr` are unchanged.

After receipt, the `rcap` and `rvr` indicate, respectively, the number of received capabilities and received virtual registers.

# Chapter 6

# Sender's Capabilities

Coyotos is an extensible capability system. New objects are introduced by **Sender's Capabilities**. There are sender's capabilities for FCRBs, receive queues, and wrapper objects. Where an FCRB, receive queue, or wrapper capability provides operations on the corresponding kernel abstraction, a sender's capability conveys authority to send a message to an object that is implemented by an application.

A particularly important feature of sender's capabilities is the presence of a *protected payload* field. This is a field stored in the capability that is delivered to the implementing object server. The protected payload field value is set when the sender's capability is fabricated, and can neither be observed nor modified by the invoking client.

## 6.1 FCRB Sender's Capability

When an FCRB sender's capability is invoked, the sequence of processing is:

1. The payload matching (PM) test is performed. If PM=1 and the protected payload from the sender's capability does *not* match the protected payload stored in the FCRB, an invalid capability exception is reported to the invoker and delivery processing terminates.
2. If FCRB is stateful (ID=0):
   (a) If the FCRB is blocked (BL=1) and the send is blocking, enqueue the sender on the FCRB until the FCRB becomes unblocked. Delivery processing terminates.
   (b) If the FCRB is blocked (BL=1) and the send is non-blocking, operation completes successfully with sent payload discarded. Delivery processing terminates.
   (c) If the FCRB is unblocked (BL=0):
      i. The protected payload of the sender's capability is copied to the protected payload field of the FCRB.
      ii. The first two transmitted registers are copied to W0 and W1, respectively.
      iii. If the receive buffer accepts a long message (MS=0), the balance of the message is copied according to the limitations imposed by the extended receive descriptor. The mechanism of this transfer is described separately in the chapter in the chapter on Capability Invocation. If the send is non-blocking, the sent payload will be truncated without generating an error in the event of memory faults or blocking.
      iv. The BL bit is set (BL := 1).
3. If delivery has not terminated:
   - The FCRB is enqueued for delivery on the receiving process if it is not already enqueued.
   - An FCRB arrival event is posted.

Note that if MS=1, the balance of the extended receive descriptor field is uninterpreted by the message transfer mechanism, and may be used to store application-defined data.

## 6.2 Multithreaded Servers

Scheduler activation-based systems can make use of at most one kernel thread per hardware CPU in a given application. User-level multithreading can be combined with use of multiple FCRBs to provide nealy all of the benefits of kernel multithreading. When multiple hardware processors are present, avoiding unnecessary data motion across CPUs is essential to overall system performance. By the time data is delivered to the recipient cache, it is too late to efficiently select another processor without significant bus overheads. Receiver CPU demultiplexing must therefore be performed within the kernel.

To support this kernel-level CPU demultiplexing, Coyotos provides kernel-implemented **receive queues**. An unblocked FCRB that designates a receive queue in its `rcvQueue` field is enqueued on that receive queue at the time of unblock (when BL goes to zero) and dequeued at time of block (when BL goes to 1). Such an FCRB may be invoked either by a sender's capability to the FCRB or by invocation of a sender's capability to the receive queue.

When a receive queue sender's capability is invoked, the processing sequence is as follows:

1. If no FCRB is enqueued on the receive queue:
   - If the send operation permits blocking, the sender is enqueued on the receive queue.
   - Otherwise, the message is silently discarded.
2. Otherwise:
   (a) Kernel selects an enqueued FCRB according to machine-dependent selection criteria.
   (b) Execution proceeds as though a sender's capability to this FCRB had been invoked, where the protected payload of the hypothetically invoked FCRB sender capability is determined by taking the protected payload of the actually invoked receive queue sender's capability.

The net effect of this mechanism is to add an additional level of indirection that is used for CPU demultiplexing.

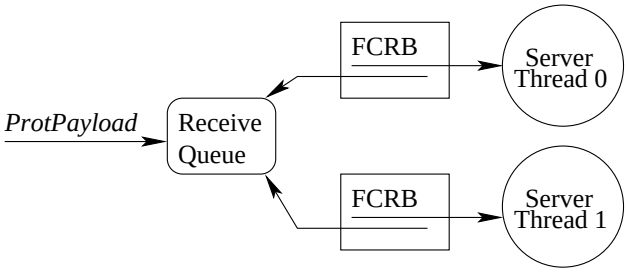


Figure 6.1: Receive queue arrangement

## 6.3 Selective Revocation

The classic challenge of capability systems is to provide selective revocation of capabilities. Coyotos implements the "caretaker" pattern proposed by Redell [19] in the form of a `wrapper` object. A wrapper wraps a single capability. The control interface of a wrapper allows the contained capability to be replaced. The sender's capability allows the wrapper to be invoked.

A server wishing to provide selective revocation of objects that it implements should implement a single FCRB for open waits, which is in turn named by multiple wrappers:

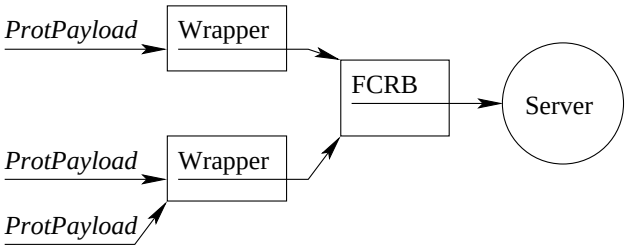


Figure 6.2: Selectively revocable client capabilities

Wrappers may also be used to wrap receive queue capabilities.

When a sender's capability to a wrapper is invoked, execution proceeds as though a sender's capability to the wrapped object had been invoked, where the protected payload of the hypothetically invoked sender's capability is determined by taking the protected payload of the actually invoked wrapper sender's capability.

## 6.4 Rationale

This section is explanatory and non-normative. It describes the intended use of the mechanisms above so that the role of the FCRB may be less confusing.

### 6.4.1 Payload Matching

In order to correctly implement remote procedure calls, the underlying messaging transport must provide some means to ensure that unexpected replies are not injected. This may take the form of a closed wait, as in L4 [9] or a specialized waiting state [13]. In addition, some means is needed to differentiate among the interfaces that a service implements. In Coyotos, the protected payload mechanism combined with the payload matching check serves both requirements.

In the case of a server implementing multiple interfaces or objects, the protected payload field can be used to identify the interface or object invoked. In this case, the protected payload is assigned over an extended period of time. The server receives with payload matching disabled, and is advised of the protected payload from the capability that was invoked.

In the case of a caller expecting a reply to a particular capability, the protected payload field of the FCRB is advanced before each call. This has the effect of establishing a session that is terminated at the end of each call. When the initial reply is sent, the BL field is set (BL:=1), causing further sends to block or be discarded. The caller can then reuse the reply FCRB by incrementing its protected payload field. There is direct support for this in the `InvokeCapability` operation.

### 6.4.2 Idempotent Messages

An idempotent FCRB (ID=1) supports non-blocking notification. In particular, it eliminates the need for the helper threads described in [10]. The recipient can use the payload matching feature to allow the FCRB to be serially reused. If payload matching is not used, the receiver cannot in general determine who sent the idempotent message, because the protected field is not updated during send.

**Note**

There has been some discussion about implementing a variant of this idea that merges (presumably by bitwise-OR) the values of W0 and W1. This would provide direct support for signals. No decision has been made yet concerning this idea.

### 6.4.3 Stateful Messages

Stateful messages (ID=0) are the normal kind associated with IPC systems in other microkernels. The payload of stateful messages is described in the chapter on Invocation.

## 6.5 Protected Payloads Under Composition

Through combinations of wrappers, receive queues, and FCRBs, it is possible to create arrangements in which multiple protected payloads may appear between a client and the server. In the absence of specification, a client might wrap an existing capability with a wrapper in order to alter the protected payload received by the server.

The disambiguating rule is that the server receives the *last* protected payload traversed on the path of capabilities from client to server. This ensures that once a server implements a sender's capability, nothing done by a client can cause the protected payload to be altered.

# 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 capablities. 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 register interest in hardware interrupt events. 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.

The `ioperm` capability describes the I/O address space in which all I/O ports implemented by the hardware are readable and writable to the application.

# Part II

# Microkernel Interfaces

# 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.

## Constants

| Name | Type | Value | Description |
|---|---|---|---|
| **msgLimit** | uint32 | 65536 | Maximum number of bytes that a message may contain. |

## Exceptions

**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. In theory, this exception should not be possible if the invocation was performed using CapIDL-generated stub code.

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.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.

## 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.<br>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. |
| **noExecute** | uint32 | 2 | Capability does not permit execution.<br>On hardware that supports a non-execute control bit, attempts to execute from a range marked `noExecute` will generate exceptions. |
| **readOnly** | uint32 | 4 | Capability is read only.<br>This capability does not permit mutation of the target object. |

## Operations

**reduce** Return copy of current memory capability with reduced permissions.

```
memory reduce(
  uint32 mask);
```

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

**getRestrictions** Return the restriction bits set for this memory capability.

```
uint32 getRestrictions();
```

# coyotos.page

**Interface** `coyotos.page`

**Derivation:**

```
coyotos.cap
  coyotos.memory
    coyotos.page
```

**Synopsis:** Page interface

## Operations

**erase** Zero this page.

```
void erase();
```

**Raises:** `NoAccess`

**copyFrom** Copy content of another page.

```
void copyFrom(
  page other);
```

**Raises:** `NoAccessRequestError`

# coyotos.capage

**Interface** coyotos.capage

**Derivation:**

```
coyotos.cap
  coyotos.memory
    coyotos.capage
```

**Synopsis:** Capability page interface

## Operations

**erase** Rewrite this capage to null capabilities.

void erase();

**Raises:** NoAccess

**copyFrom** Copy content of another capage.

void copyFrom(
  capage other);

**Raises:** NoAccessRequestError

# coyotos.opatt

**Interface** coyotos.opatt

**Derivation:**

```
coyotos.cap
  coyotos.memory
    coyotos.opatt
```

**Synopsis:** Opaque Patt interface.

Primary description of PATTs can be found in the documentation for coyotos.patt.

An opaque PATT can be used for address references and fault delivery, but its slots can neither be fetched nor stored.

## Enumerations

**accessType** access types for fault propagation:

| Name | Type | Value |
|---|---|---|
| **read** | uint32 | 1 |
| **write** | uint32 | 2 |
| **execute** | uint32 | 3 |

## Operations

**probe** Verify an address for a given access type.

```
bool probe(
  coyaddr_t addr
  uint32 access
  bool andFault);
```

**Raises:** NoAccess

Traverses the address addr (relative to the current PATT), and attempts to validate the address for the access type(s) specified by access, which should be the result of ORing values from the accessType enum above. If andFault is true, the nearest enclosing keeper will be invoked in order to validate the address if needed and possible.

Note that if the keeper is invoked, the reply will come from the keeper rather than the kernel.

# coyotos.patt

**Interface** coyotos.patt

**Derivation:**

```
coyotos.cap
  coyotos.memory
    coyotos.opatt
      coyotos.patt
```

**Synopsis:** Prefixed Address Translation Table

A prefixed address translation table (PATT) is the mechanism for composing address spaces. The data structure and basic idea of PATTs 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** | int32 | 16 | number of slots in a PATT |

## Exceptions

**NoSpace** Mapping insertion failed for lack of free slots.

## Operations

**insert** Insert associative entry for cap c with prefix pattern.

```
void insert(
  pattern_t pattern
  cap c);
```

**Raises:** NoSpaceNoAccess

Inserts a new (prefix, capability) pair into the PATT. Raises NoSpace if all slots of the PATT are already full. If the specified prefix already exists, the associated capability is overwritten.

**fetch** Fetch capability associated with the given pattern.

```
cap fetch(
  pattern_t pattern);
```

**drop** Deletes any (pattern, capability) mapping whose value matches addresses between `base` and `bound`.

```
void drop(
  coyaddr_t base
  coyaddr_t bound);
```

**getPatterns** Return an array containing all valid patterns in this `PATT`.

```
anon0 getPatterns();
```

**getHeight** Fetch the height `h` and the residual bits `r` for this `PATT`.

```
void getHeight(
  OUT uint32 h
  OUT uint32 r);
```

**setHeight** Set the height `h` and the residual bits `r` for this `PATT`.

```
void setHeight(
  uint32 h
  uint32 r);
```

**Raises:** `NoAccessRequestError`

Note that changing the `h` and `r` values may lead to unmatchable patterns if `r` grows or `h` shrinks. It is the application's responsibility to rearrange the pattern values.

The kernel imposes the invariant that $h > r$, and also that $r >= \log_2(pagesize)$. Attempts to perform updates that would violate these constraints will raise a `RequestError` exception.

**constrainPatterns** Constrain all pattern values to have non-zero bits only in bit positions that are (a) within the specified [`r.h`] interval or (b) within the page offset positions.

```
bool constrainPatterns(
  uint32 r
  uint32 h);
```

Returns false and leaves `PATT` unmodified if the result of the operation would violate the disjoint pattern requirement. Returns true if the operation is successful.

**copyFrom** Copy those entries of another `PATT` that would match addresses between `base` and `bound`.

```
void copyFrom(
  patt other
  coyaddr_t base
  coyaddr_t bound);
```

**Raises:** `NoAccessNoSpaceRequestError`

**getOpaquePatt** Returns an opaque `PATT` capability to the same `PATT`.

```
opatt getOpaquePatt();
```

**makeWindow** Make a new background window capability.

```
void makeWindow(
  pattern_t pattern
  coyaddr_t offset
  uint32 h
  uint32 mask);
```

**Raises:** `NoSpaceNoAccessRequestError`

The `offset` argument specifies the offset *relative to the background space* that this window names. The `mask` argument gives the permissions conveyed by the window. The `h` argument determines the size of the mapped region.

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

# coyotos.window

**Interface** coyotos.window

**Derivation:**

```
coyotos.cap
  coyotos.memory
    coyotos.window
```

**Synopsis:** Address space window interface

## Operations

**getOffset** Retrieve the offset value stored in this window capability.

coyaddr_t getOffset();

# 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.

## 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) |
| **ia64** | uint32 | 1 | Intel `ITANIC` |
| **amd64** | uint32 | 2 | `AMD64` |
| **sparc** | uint32 | 3 | Sun 32-bit `SPARC` |
| **sparc64** | uint32 | 4 | Sun 64-bit `SPARC` |
| **arm** | uint32 | 5 | Acorn `RISC` Machine |
| **coldfireV4** | uint32 | 6 | Later coldfire processors |

**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 above 127 are reserved for architecture-specific fault codes.

| Name | Type | Value | Description |
|---|---|---|---|
| **NoFault** | uint8 | 0 | Process currently does not have any fault. |
| **InvalidAddr** | uint8 | 1 | Issued address was undefined. |
| **Access** | uint8 | 2 | Insufficient access rights for the issued memory reference. |
| **PattLimit** | uint8 | 16 | PATT traversal limit exceeded. |
| **MalformedSpace** | uint8 | 17 | Address space PATT arrangement is malformed, or PATT contains an inappropriate capability type. |
| **MalformedProcess** | uint8 | 32 | Process is malformed. |
| **NoAddrSpace** | uint8 | 33 | Process has invalid/maltyped address space capability. |
| **NoSchedule** | uint8 | 34 | Process has invalid/maltyped schedule capability. |
| **BreakPoint** | uint8 | 35 | Process encountered a breakpoint instruction (PC=&bpt).<br>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 | 36 | Process encountered a breakpoint instruction (PC=&bpt+1).<br>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. |
| **BadOpcode** | uint8 | 37 | Process issued an illegal or unknown instruction. |
| **Alien** | uint8 | 38 | Process marked as “alien” performed an invocation instruction. |
| **DivZero** | uint8 | 39 | Process performed an integer divide by zero |
| **BadAlign** | uint8 | 40 | Process performed a checked misaligned memory reference. |
| **NoFPU** | uint8 | 41 | No floating point unit available.<br>This exception indicates that there is neither a hardware floating point unit nor a kernel-provided software emulation available on this machine. |
| **FPfault** | uint8 | 42 | Unsupported floating point instruction.<br>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 | 43 | Debug Exception<br>A hardware debugging event has occurred. |
| **Overflow** | uint8 | 44 | Overflow trap. |
| **Bounds** | uint8 | 45 | Bounds Violation. |
| **GeneralProtection** | uint8 | 46 | General Protection fault. |
| **StackSeg** | uint8 | 47 | Stack Segment fault. |
| **SegNotPresent** | uint8 | 48 | Segment Not Present fault. |
| **SIMDfp** | uint8 | 49 | SIMD floating point error. |

**cslot** Capability slots of a process.

**This is obsolete**

| Name | Type | Value | Description |
|---|---|---|---|
| **handler** | uint32 | 0 | Fault handler. |
| **dataSpace** | uint32 | 1 | Data address space. |
| **capSpace** | uint32 | 2 | Capability address space. |
| **ioSpace** | uint32 | 3 | I/O address space. |
| **schedule** | uint32 | 4 | Schedule. |
| **upcbPage** | uint32 | 5 | `UPCB` page |
| **capRegs** | uint32 | 6 | Capability registers page |

## Structures

**FaultInfo** Canonical fault information structure.

```
struct FaultInfo{
  FC      code;   Fault code.
  uint64  info;   Auxiliary fault information.
};
```

The `FaultInfo` structure is used to describe the current fault status of this process. The info field most often holds an address. A canonicalized fault address will always be expressed as a *software* address, which is 64 bits regardless of the hardware architecture.

## Operations

**stop** Stop the process.

```
void stop();
```

**Raises:** `NoAccess`

Transitions the process to the idle state. When idle, a process will not attempt to initiate new instructions.

**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`.

**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`

# coyotos.fcrb

**Interface** coyotos.fcrb

**Derivation:**

coyotos.cap
  **coyotos.fcrb**

**Synopsis:** FCRB interface

An FCRB is the encapsulation of a receive buffer for a pending message. FCRBs are described in detail in the Coyotos Microkernel Specification.

## Operations

**setProcess** Set the recipient process .

```
void setProcess(
  coyotos.process p
  coyaddr_t erb);
```

**setPayload** Set the protected payload value l and payload matching requirements m.

```
void setPayload(
  payload_t pl
  bool pm);
```

**setERB** Set the extended receive descriptor pointer *rb* for this FCRB.

```
void setERB(
  coyaddr_t erb);
```

**setBlocking** Set or clear the blocked message bit.

```
void setBlocking(
  bool bl);
```

**setIdempotent** Set or clear the idempotent bit id.

```
void setIdempotent(
  bool id);
```

**setReceiveQueue** Set the receive queue associated with this FCRB

```
void setReceiveQueue(
  coyotos.rcvqueue rq);
```

**makeSendCap** Fabricate a sender's capability to this FCRB.

```
cap makeSendCap(
  payload_t payload);
```

# coyotos.rcvqueue

**Interface** coyotos.rcvqueue

**Derivation:**

coyotos.cap
**coyotos.rcvqueue**

**Synopsis:** Receive queue interface.

## Operations

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

```
bool dequeue(
  OUT coyotos.fcrb fcrb);
```

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

```
cap makeSendCap(
  payload_t payload);
```

# coyotos.wrapper

**Interface** `coyotos.wrapper`

**Derivation:**

```
coyotos.cap
  coyotos.wrapper
```

**Synopsis:** Wrapper interface.

## Operations

**setForwardingCap** Sets the forwarding capability, which must be an FCRB or receive queue capability.

```
void setForwardingCap(
  coyotos.cap cap);
```

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

```
cap makeSendCap(
  payload_t payload);
```

# coyotos.fault

**Interface** coyotos.fault

**Derivation:**

coyotos.cap
  **coyotos.fault**

**Synopsis:** Interface implemented by fault handlers.

This is the interface that must be implemented by fault handlers. The interface definition for this interface is determined by the kernel.

## Enumerations

**accessType** access types for fault propagation:

| Name | Type | Value |
|---|---|---|
| **read** | uint32 | 1 |
| **write** | uint32 | 2 |
| **execute** | uint32 | 3 |

## Operations

**raise** Raise a kernel-generated exception.

```
void raise(
  uint32 FC
  uint64 auxInfo);
```

Takes the exception ID in exNo, and a single word of auxiliary information in auxInfo.

# 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.keybits

**Interface** coyotos.keybits

**Derivation:**

coyotos.cap
  **coyotos.keybits**

**Synopsis:** Kernel interface to key representation.

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

## Constants

| Name | Type | Value | Description |
|---|---|---|---|
| **VERSION** | uint32 | 1 | Current kernel-implemented keybits version.<br>This version number represents the keybits version returned by the kernel as of the time this interface was last updated. The actual layout of the four word array matches the capability layout as shown in the Coyotos Microkernel Reference. All reserved fields are reported as zero. The hazard bit is always reported as zero. |

## Structures

**info** Capability as data response structure.

```
struct info{
   uint32   version;
   bool     valid;
   anon0    w;
};
```

## Operations

**get** void get(
  cap c
  OUT info bits);

Return the representation of a key as data.

Open Issue: should the get() interface accept as an argument the desired keybits version, or is the version number output only?

# 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 |
|---|---|---|
| **clVoid** | uint32 | 0 |
| **clWindow** | uint32 | 1 |
| **clMemory** | uint32 | 2 |
| **clSched** | uint32 | 3 |
| **clExtended** | uint32 | 4 |
| **clOther** | uint32 | 255 |

## Operations

**classify** Return the classification of the passed capability.

```
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.

**compare** Compare two capabilities for (exact) identity.

```
bool compare(
  cap c1
  cap c2);
```

# coyotos.irqctl

**Interface** coyotos.irqctl

**Derivation:**

coyotos.cap
  **coyotos.irqctl**

**Synopsis:** Low-level interrupt control capability.

An irqctl capability provides the authority to be notified when a given interrupt to occur. The kernel implements an internal receive queue associated with each hardware interrupt. When the interrupt arrives, all FCRB's posted on that interface will be notified of arrival by means of a message send.

**Open Issue:**

Exactly one of getInterruptQueue or registerFCRB should be implemented. Which one will depend on what makes more sense when this part of the kernel is actually implemented. The issue is that it may not be straightforward to get an OID reliably assigned to an interrupt line. If it *is* possible to get that association working the getInterruptQueue interface is preferred because it gives finer control over who can wait on what.

Note, however, that if fcrb ever implements a getReceiveQueue operation the IRQ receive queue will be exposed in any case, so resolving this issue may turn out not to be optional in any case.

## Operations

**getInterruptQueue** Retrieve a capability to the receive queue associated with the specified hardware interrupt.

```
coyotos.rcvqueue getInterruptQueue(
  uint32 irq);
```

**registerFCRB** Register an FCRB to be notified when this interrupt arrives. If the FCRB is already bound to a receive queue, it will be unbound from the existing receive queue and associated with the per-IRQ queue.

```
void registerFCRB(
  uint32 irq
  coyotos.fcrb fcrb);
```

# 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.ioperm

**Interface** `coyotos.ioperm`

**Derivation:**

```
coyotos.cap
  coyotos.ioperm
```

**Synopsis:** I/O Permissions capability

A process having the ioperm capability is permitted to execute I/O instructions from user level. The current Coyotos interface does not provide a mechanism to authorize individual I/O ports at the kernel layer. This is a deficiency that needs to be corrected.

There are no operations specific to ioperm capabilities.

# coyotos.sleep

**Interface** `coyotos.sleep`

**Derivation:**

```
coyotos.cap
  coyotos.sleep
```

**Synopsis:** Kernel delay mechanism.

The sleep capability allows a process to post an `FCRB` that will be received at a specified later time. The invoking process need not block for the reply, so the same interface can be used to post watchdog timers.

## Operations

**sleep** Sleep until the specified number of milliseconds has passed or the system is rebooted.

```
void sleep(
  uint64 ms);
```

**sleepTill** Sleep until the specified number of milliseconds from epoch or until the system is rebooted.

```
void sleepTill(
  uint64 ms);
```

# 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.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.

# Appendix A

# Change History

This section is an attempt to track the changes to this document by hand. It may not always be accurate!

## A.1 Changes in version 0.1

- First document version.

## A.2 Changes in version 0.2

- Introduction of asynchronous IPC, with associated new objects and revision of fault handling model.

# Bibliography


[1] 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.

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

[3] J. S. Shapiro. "Vulnerabilities in Synchronous IPC Design" *Proc. 2003 IEEE Symposium on Security and Privacy*. 2003.Oakland, CA, USA.

[4] 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.

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

[6] Charles Landau. "CapROS: The Capability-based Reliable Operating System." `http://www.capros.org`.

[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] 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] 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] T. E. Anderson and B. N. Bershad and E. D. Lazowska and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism" *Proc. 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, USA. pp. 95–109. 1991.

[16] B. D. Marsh and M. L. Scott and T. J. LeBlank and E. P. Markatos, "First-Class User-Level Threads" *Proc 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, USA. pp. 110–121. 1991.

[17] 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.

[18] 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.

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

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

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