

Differences Between Coyotos and EROS — A Quick Summary[†]

Version 0.3, revised 2 April 2006

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

May 12, 2005

Abstract

Coyotos is a (mostly) evolutionary step from the EROS kernel. This document briefly summarizes the major architectural differences between EROS and Coyotos, and gives the rationale for each change. It looks both at the changes that have been definitely decided, and also at some questions in the Coyotos architecture that still remain open.

1 Overview

Coyotos, the successor to EROS [3], has three main objectives:

- Correct some known deficiencies relative to the EROS architecture that impede performance or increase complexity.

4/2/2006 Update: When this document was first written, the design and evaluation of solutions was a research issue. That work is still underway, but the resulting design is now stable modulo minor tweaks that may become motivated during development. The kernel specification can now be found online at the website.

- Explore the use of software verification techniques to achieve higher confidence in the correctness and security of the kernel and the key system components (a topic for another document). This is a *research* objective.
- Re-examine whether persistence was a good idea.

4/2/2006 Update: The answer is now clearly *yes*.

We are sticking firmly with the idea of capability-based systems. For the most part, the goal in Coyotos is refinement rather than replacement of ideas.

The most important changes relative to the EROS architecture are conceptually minor and not particularly controversial. A few are controversial, but possibly important as Coyotos moves forward. One or two should definitely be considered exploratory in nature. Overall, the basic theory of operation for the Coyotos system is very similar to the EROS theory of operation, and this should allow us to preserve most of the design elements of EROS applications — notably `capidl` — and much of the structural design of the EROS kernel

I Definite Changes

2 Address Spaces

EROS (and KeyKOS [1] before it) both used the **node** data structure to describe memory mappings. A node is a software-defined hierarchical page table structure. The kernel constructs the real hardware tables by traversing the node structures.

2.1 EROS vs. KeyKOS

There were only minor differences between EROS and KeyKOS in the design of the mapping mechanism :

- KeyKOS nodes held 16 capabilities, EROS expanded this (for reasons having little to do with memory mapping) to 32 slots.
- KeyKOS had two types of node capabilities: “node” and “segment”. The main difference between these is that a segment capability is opaque — you can use

[†] Copyright © 2005, Jonathan S. Shapiro.

it in a memory context to define a mapping, but you cannot directly fetch capabilities from a node using a segment capability. In EROS, the segment capability was renamed an “address space capability.”

- KeyKOS had something called a “red segment,” which was a rather odd sort of node that carried optional additional information such as a keeper capability, a background address space, and so forth. In practice, the primary use of red segment nodes were to specify keepers and growable address spaces, but red segments had a secondary use as a revocable front end to a start capability. KeyKOS space bank capabilities, for example, were really red segment capabilities that wrapped a start capability to the actual space bank.

Because red segments had important uses outside of memory contexts, EROS eventually replaced this with a simplified form called a “wrapper node.” The wrapper node largely served the same function as the red segment node, but its specification was significantly simpler. Also, wrappers added a “replace data register” option that we found very useful in a number of applications.

2.2 Problems With Nodes

In the KeyKOS and EROS systems, the Node data structure suffers from several problems.

Overuse Nodes were reused for too many purposes. In addition to their role in the memory system, nodes were also the underlying storage resource for processes. This led to considerable kernel complexity. Nodes simply aren’t a very pleasant way of describing process state.

Representational Inefficiency A 16-ary or 32-ary hierarchical mapping structure needs too many levels to efficiently describe an address space, particularly when the address space consists of sparse, dense sequences of pages. While Nodes provided a self-encoding height, this was not as useful in reducing the size of the mapping tree as we would have liked.

Further, the encoding of Nodes requires a surprisingly complicated traversal algorithm to deal with many cases. Perhaps Charlie Landau will find a simpler one in CapROS [2], but I am skeptical.

Size Partitioning Because EROS and KeyKOS are persistent, both systems require that allocate nodes and pages separately on the disk. Two classes of disk-based objects is certainly better than three, but one would be very much better than two. Coyotos will use a different resource allocation mechanism, allowing all persistence to be accomplished in terms of pages.

Height Encoding . The height of a node was (in my opinion) mis-encoded. It should have been an attribute of the node itself rather than an attribute of the capability naming the node. As an additional, minor issue, it would have been better for a variety of reasons to encode the height using \log_2 of the address space size in bytes rather than \log_{16}/\log_{32} of the address space size in pages.

2.3 Changes in Coyotos

The memory mapping subsystem is being completely replaced in Coyotos.

Coyotos does not have a Node structure. Processes are first-class objects, and a replacement data structure is being used in the memory tree: the PATT. The PATT structure can be seen as a generalization of the Node structure, but its encoding has been completely reworked.

There is only one type of PATT capability. The distinction analogous to the difference between “address space” capabilities and “node capabilities” is subsumed by a new permission bit in the capability permissions mask: the **opaque** bit. The EROS implementation has been moving in this direction, but the implementation was never completed.

A PATT (Prefixed Address Translation Tree) consists of:

<i>Height</i>	The maximal size in bytes of the space described by this PATT, expressed as $\log_2(\text{SizeInBytes})$.
<i>Residual</i>	The number of bits in the address that will remain to be translated <i>after</i> this PATT has been successfully traversed by the translation algorithm.
<i>Slots</i>	A 16 element array of pairs of the form (<i>prefix, capability</i>). Matchable prefix values must be disjoint.

The important innovation in the PATT structure is that there are no bits used for indexing purposes. Every slot has an associated prefix, and the prefix must match the incoming address when bits in the residual portion of the incoming address are disregarded. The effect of this is that any PATT can describe a sparse space of arbitrary size 2^h having up to 16 subspaces within it. A subspace can either be a PATT describing some smaller space or a page.

Translation against a PATT proceeds by temporarily masking out the residual bits of the address, and then doing a sequential, first-match comparison against the prefix values. Because the least mappable unit is a page, there are low-order bits in the prefix that must be zero in order to match successfully (because the page offset bits are *always* part of the residual bits). An invalid slot is indicated

by setting the least significant bit of the prefix to 1. Specification of keepers and background spaces¹ similarly are encoded by using distinguished prefix values.

The resulting data structure is significantly simpler to traverse during mapping construction, preserves all of the clever inverse mapping encoding tricks that were feasible in EROS/KeyKOS, and requires fewer levels and fewer total count of PATTs for most address spaces.

3 Address Translation

Both EROS and KeyKOS rely on a data structure known as the **depend table**. Ignoring various compression tricks, this structure stores a pair of the form (capability slot address, hardware PTE address). The purpose of the depend table is to allow page table entries to be invalidated whenever a memory capability slot is overwritten.

In hindsight, the depend table was unnecessarily inefficient. A better design would store the address of the Node (rather than the slot), the address of the *starting* PTE slot potentially associated with that node, and a scaling factor describing how many PTEs are associated with each slot of that Node.

For PATTs, matters are somewhat more complicated, and the corresponding dependency structure must adapt. In Coyotos, the dependency structure will need to be adapted to reflect the prefix-based nature of PATTs. One bit of good news here is that the number of depend table entries required should be dramatically reduced.

10/27/2005 Update: Eric Northup is close to completing his writeup of the analysis of PATTs, and we will have a more complete description of this issue when that is complete.

4/2/2006 Update: Eric Northup's design has now been reviewed, and the differences are not as complicated as we had thought. We now know how to do the dependency tracking, and also how to do efficient translation. Eric's implementation will require some transformation, but the basic structure of his approach should be directly translatable into the Coyotos implementation.

4 Processes

There have been many changes in Coyotos to the Process abstraction. Most of these changes are individually minor,

¹ It is not clear at this time whether Coyotos will preserve background spaces in the architecture, as no compelling use case for them was ever found in EROS. We *will* reserve a prefix code point for them should they ever prove useful in the future.

though some have significant implications for implementations.

4.1 Processes are First Class

Both KeyKOS and EROS represented process state as an arrangement of Nodes. The details between the two systems varied in only minor ways. While it would be possible to use the same style of design in Coyotos, we have decided instead to make processes into first-class objects. There are several reasons to do this:

Elimination of Consistency Issues Making processes first class eliminates the need for consistency management between the "process as node" view of the world and the "process as process" view of the world, which was a significant source of complexity in the EROS and KeyKOS implementations. EROS had already started to move in this direction by revising the process capability interface so as to make most node references unnecessary.

Note that this significantly improves the performance of IPC, and simplifies both the process dispatch and the register save logic of the kernel.

Impact of Endpoints In KeyKOS and EROS, the means for invoking a process was the start capability, which directly named the process. For reasons described below, start and resume capabilities have been replaced by endpoints and endpoint capabilities in EROS. One consequence of this is that processes are not pageable.² In the absence of process paging, there is no strong rationale to express the process state using a disk-based data structure, and there are many reasons to want to *avoid* doing so.

4/2/2006 Update: The endpoint idea did not survive contact with the enemy, and was subsequently replaced by first-class receive buffers (FCRBs) and scheduler activations. Readers are encouraged to examine the description of FCRBs in the Coyotos kernel specification to learn about these.

Out of Kernel Helpers Coyotos follows the trend in later versions of EROS to significantly relax one of the kernel design rules. In EROS/KeyKOS, the kernel did not rely on the correct behavior of application code for its correct execution — not even *trusted* application code. Coyotos adopts the view that certain portions of the fundamental system logic can and should be implemented in trusted processes that have the same standing as the kernel with respect to enforcement of security.

A consequence of this is that all responsibility for disk-based object representations can be offloaded to user-

² More precisely, processes are not pageable in the first version of Coyotos. Conceptually, we know how to implement process paging. We are deferring the implementation to a second, backwards compatible iteration on the kernel interface.

mode code, where it can be debugged.

4.2 Changes to Process State

Coyotos makes a number of changes to per-process state relative to EROS and KeyKOS.

Capability Address Spaces The KeyKOS and EROS kernels both implemented per-process “capability registers.” In Coyotos, we have decided instead to implement capability address spaces and adapt the capability invocation logic to use addresses relative to the capability address space rather than register numbers. The new implementation *may* prove to be somewhat less space-efficient than the old one for small processes, but this seems unlikely. Two key reasons for introducing a capability address space are the need for some applications to easily manage larger numbers of capabilities, and the need to implement some form of capability stack so that libraries can manipulate temporary capabilities. Neither of these was particularly efficient in either EROS or KeyKOS. In EROS, we found this to be a pragmatically important performance problem for certain key applications.

Virtual Registers Every coyotos process has a page of “virtual registers.” These are used to describe capability invocations, and also to store the data payload for small invocations. The virtual register page is optionally mapped into the process address space. Processes that do not map a virtual register page cannot, as a practical matter, perform capability invocations.

As a corollary to this change, all kernel-implemented capabilities in Coyotos are designed to return their results the virtual registers, which eliminates much of the string processing complexity of the kernel invocation path. The kernel is entitled to rely on the presence of the virtual registers page in any process that is executing, and does not need to perform any validity checks on the virtual registers page. The only string manipulation in the Coyotos kernel is done in the interprocess communication path.

4.3 Fault Handling

Both EROS and KeyKOS handled exception processing using a distinguished subtype of resume capability called a **restart capability**. Coyotos does not have either start or resume capabilities, and the restart capability has been replaced with a distinguished subtype of process capabilities. Resumption of a faulted process is accomplished by invoking this capability.

5 IPC

The most significant committed change in Coyotos is in the area of interprocess communication (IPC). This section addresses conceptual differences in the interprocess communication abstraction. There are also generic changes to the capability invocation mechanism that are discussed below.

Endpoints By far the most important change in the IPC design is the introduction of endpoints. An **endpoint** is a new kernel object type that encapsulates a rendezvous queue for interprocess communication. In order to send a message to a process, the sending process must perform a send operation on an endpoint descriptor. In order to receive from an endpoint, the receiving process must perform a receive operation on an endpoint descriptor. Endpoint descriptors carry permission bits that indicate which operations (send, receive, destroy) on a given endpoint are permitted for each descriptor.

The main motivation for moving endpoints into the kernel was the desire to do thread demultiplexing within the IPC path. Orran Krieger pointed out that it is extremely inefficient to do user-level thread dispatch on a multiprocessor, because data ends up in the wrong L2 CPU cache. However, multiple receive endpoints can also be used to selectively classify senders into different service classes.

The receiver on an endpoint receives both an endpoint-specific value (to a reserved virtual register) and a value that is specific to the endpoint capability invoked in the send operation (replacing the KeyKOS/EROS key data field).

There is no capability type in Coyotos analogous to a resume capability. In the typical case, a calling process will have an endpoint that it uses to receive all incoming messages. This endpoint can be revoked explicitly at the discretion of the incoming process, causing outstanding endpoint capabilities (including send capabilities) to become invalid.

FCRBs (*Added 4/2/2006*) The endpoint idea did not play out, and was dropped from the architecture. We decided to move to a variant of a “scheduler activations” design. This allows message payload to be delivered even when the receiving process is not explicitly blocked, and allows a tighter meld between user-mode and kernel-mode thread scheduling. The new mechanism is described in the FCRBs section of the Coyotos kernel specification, along with the surrounding discussion of process states, event delivery, and the meld of activations with messaging.

Events One of the problems we encountered in the implementation of the EROS networking stack [4] was an asynchronous rendezvous problem. When two processes are communicating using shared memory, it is helpful to have some atomic event mechanism in which (a) the

sender never blocks, but (b) the receiver is not preempted. Think of this as comparable to signals, but without the preemption. The measured effect of *not* having this mechanism in the network stack was a 15% reduction in achievable network bandwidth.

Coyotos incorporates a new type of invocation that is only supported by endpoint capabilities. It allows events (represented as bits) to be posted to an endpoint. On the next receive operation, the receiver gets a distinguished message type indicating that incoming events are present.

6 Capability Invocation

Both EROS and KeyKOS have very simple IPC mechanisms. Simple is generally good, but in both systems the absence of a scatter/gather mechanism came to be an impediment. It is not entirely clear whether this was the result of a shortage of registers or truly a result of the absence of a scatter/gather mechanism. The difficulty was most obvious in the implementation of a file system, where you really wanted incoming file content to end up in a differentiable buffer from the rest of the operation's payload. Other arguments for scatter gather seem less compelling, but we may experiment with them in Coyotos.

One change that is definite is the need to revise the specification of an invocation to consist of a vector of "items" that may include capabilities from the capability address space.

Non-blocking Invocation Both the send phase and the receive phase of an invocation may specify that they are unwilling to block. This is borrowed from the L4 specification, but we do *not* provide any specific timeout value. It appears that the successor to L4 will similarly drop the ability to specify an explicit timeout.

Scatter/Gather We have written elsewhere that we intend to provide scatter/gather support in the Coyotos invocation mechanism. Recently, we realized that this may no longer be motivated. The main practical motivator for scatter/gather was the EROS file implementation. The problem there, in hindsight, is that we ran out of registers. With the introduction of virtual registers this may no longer be a problem in practice. Our current plan is to reserve a code point in the invocation mechanism to specify strings, but defer the implementation of any scatter/gather mechanism until we can see if it is still well motivated.

7 Capability GC

10/27/2005 Update: This section has been moved from "contemplated" to "definite."

One of the performance-limiting issues in EROS/KeyKOS was the implementation of prepared capabilities. In both systems, a capability to an in-memory object is placed on a circular list that is rooted at the target object. This allows efficient invalidation and conversion back to out of memory format, but it makes capability copy extremely expensive. The interacting issues are:

- Most copied capabilities are prepared, and are therefore members of a list.
- Most overwritten capabilities are prepared.
- Capability copy therefore requires unlinking the target capability (two cache misses) and then re-linking the copy (one additional cache miss). This proves to be a major cost in invocations that transfer capabilities.
- 50% of all invocations transfer at least one capability, which is that descriptor to which the reply should be transmitted.

There are two possible ways to resolve this. The first is to re-introduce capability registers into Coyotos, and rework the in-register capability format using the design for soft register rename that was described on the `eros-arch` mailing list some time ago.

The other is to change the in-memory capability format such that capability depreparation is driven by an incremental mark/sweep pass rather than by a linked list. In this design, the link chain fields would be replaced by a pointer to a secondary data structure (Figure 1). When the target object is removed or deprepared, the `MemCap` structure is updated to zero the `valid` field. Before using a capability in a given invocation, the current validity of the `valid` field in the corresponding `MemCap` structure must be checked. The `MemCap` structures are reclaimed using a mark/sweep pass, which is incrementally driven by capability preparation.

The advantage to the GC-based design is that it allows capabilities to be copied without regard to their current representation (in memory or out of memory). This eliminates all avoidable cache misses in the capability copy path.

The *disadvantages* to this approach are that (a) the incremental GC approach must be assured of convergence, and (b) in some cases the `MemCap` structure is comparable in size to the object it governs — notably indirection objects

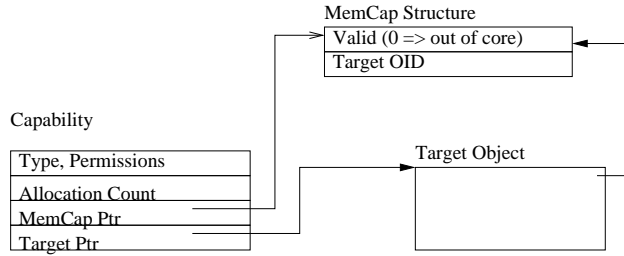


Figure 1: Alternative capability arrangement for GC.

and endpoints. There are also some less important implications for dependency tracking in the memory management subsystem.

There is some comfort in the knowledge that the old (linked) approach can be introduced as a well-understood standby and the new approach can be introduced experimentally at a later time without externally visible change.

II Contemplated Changes

8 Persistence

We are currently going back and forth on whether Coyotos will provide persistence in the style of EROS and KeyKOS. Persistence is an extremely convenient thing to have, and we need to understand what complexity it carries. The *first* implementation of Coyotos will *not* provide persistence, but the existing EROS/KeyKOS persistence design could be adapted straightforwardly to Coyotos.

Initially, our concern about persistence was the desire to deal with hard real-time applications. The introduction of endpoints has partially mitigated this problem, and one of the other changes we are contemplating may eliminate it entirely. It may be that we should now re-open the persistence question.

10/27/2005 Update: After some further thought on this issue, we have decided to keep persistence. This introduces a new design issue for Coyotos. Where EROS had only two on-disk object sizes (node and page), Coyotos has several, and we will need to rethink the design of the object store to accommodate this.

9 First-Class Kernel Heaps

With the introduction of a number of new kernel object types, it may be appropriate to re-examine the method of kernel resource allocation. The goal would be to reduce

the number of fundamental allocatable units in the system to just the page, and then come up with some construct that would allow the kernel to construct internal heaps out of these units for allocation of kernel data structures.

One proposal that we are considering is the introduction of first-class kernel heaps. Objects would be allocated from heaps rather than space banks. The role of the kernel heap is to accomplish the protected conversion of untyped page frames into objects of particular types. Kernel heaps are backed by user-mode page frame allocators, which supply the heaps with untyped frames.

One of the underlying goals in introducing kernel heaps is to introduce a layer in which everything is just a page frame, and resource allocation can be done with respect to an object of uniform size. The responsibility of the kernel heaps is to suballocate these frames as appropriate.

Kernel heaps also provide a locus from which to establish two other goals: control of persistence and control of residency. It is contemplated that kernel heaps may be marked “non-persistent,” with the meaning that no object allocated from that heap will be implicitly saved by any persistence mechanism. It is also contemplated that a kernel heap may be marked “resident,” with the meaning that all frames associated with that heap will be pinned in memory. A kernel heap that is both non-persistent and resident is suitable for use in hard real-time applications.

10/27/2005 Update: This still seems like a good idea, but it doesn’t appear to be truly essential, and we have decided to defer it in the interest of making faster progress.

III Speculative Changes

We have been going back and forth on a feature whose value and impact are uncertain: large pages. The motivation for large pages is to provide some means to deal with things like device buffers on machines whose hardware TLB and memory management subsystems can usefully exploit some form of large page size. The question is: how should large page sizes be expressed? Some of the issues:

- Should they be discovered dynamically by the memory management subsystem, or should they be explicitly allocated?
- If large pages are explicit, are they definitive? That is, is the kernel entitled to validate only some fractional portion of a large page?

This decision interacts with the notional kernel heaps design in various ways that suggest that the least hardware page size needs to be the only definitive

hardware page size.

- If large page sizes are explicit, how do they interact with issues of residency management? Might they turn out to create various ugly forms of residency pressure?
- Is a large page capability a descriptor to a page *frame*, or is it a descriptor to a page's *content*? If content, is it required that the page be contiguously stored on the disk?

All of these are currently issues under discussion, with no firm conclusions as yet.

10 Acknowledgements

Several people have made significant contributions to the design process that led to the outcome summarized here. In particular, the insight that PATTs should be fully associative is due to Eric Northup. Scott Doerrie has worked out a method for reasoning formally about the expressiveness and transformative correctness of the PATT traversal algorithm.

As always, the contributions of the KeyKOS kernel team cannot be overstated. Both EROS and Coyotos have evolved from the rather amazing work of Norm Hardy, Charlie Landau, and Bill Frantz.

References

- [1] Norman Hardy. "The KeyKOS Architecture." *Operating Systems Review*, **19**(4), October 1985, pp. 8–25.
- [2] Charles Landau. "CapROS: The Capability-based Reliable Operating System." <http://www.capros.org>.
- [3] 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.
- [4] Sinha, A., Sarat, S, Shapiro, J. S.: Network Subsystems Reloaded. *Proc. 2004 USENIX Annual Technical Conference*. Dec. 2004