

# The EROS System Structure<sup>†</sup>

Working Draft, Currently In Progress

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

January 11, 2007

## 1 Introduction

This morning Marcus Brinkmann sent a note expressing that he has some difficulty keeping the layering and structure of the Coyotos system straight in his head:

*One thing that causes me problems in these discussions ... is that I often don't know if you are talking about just the kernel or also the user space system when referring to Coyotos. The space bank in particular is logically outside the kernel and part of the user space operating system. But there seem to be some complicated interactions between the kernel design and the way memory is organized in the space bank that I likely do not fully appreciate and which seem to conflate the issues.*

As I composed what turned into a long email, I realized that this ought to be captured in linear form somewhere, so I decided to respond with this note instead.

This note provides a bottom-up view. Each “part” corresponds to a layer in the system dependency structure.

I need to acknowledge here at the start that much of this design is not mine. EROS builds on KeyKOS, whose architecture at this level is primarily the work of Norm Hardy, Charlie Landau, and Bill Frantz. In EROS, my contributions were primarily in performance, some simplification, further micro-kernelization, and formalization. Coyotos, in contrast, is a fairly different beast.

### 1.1 Origins and Confusions

The Coyotos system structure is expected to be very similar to the EROS structure, which is what this note de-

scribes. EROS [2] is in turn architecturally derived from KeyKOS [1]. Information on EROS can be found on the web at [www.eros-os.org](http://www.eros-os.org). Information on KeyKOS lives at <http://www.cis.upenn.edu/~KeyKOS>. Both of the cited papers are very dense. The KeyKOS paper is *notoriously* dense.

Several attributes make the Coyotos/EROS system structure hard to keep straight:

- From the application code perspective, the system is just a collection of objects. It happens that some of these objects are implemented by the kernel and that some other objects are relied on to provide (or enforce) some of the system's guarantees. It often turns out that the final guarantee emerges as much from the *arrangement* of pieces as the pieces themselves, so it isn't always obvious which piece does what.

As a human matter the “collection of objects” view leads developers to be intentionally non-specific about which pieces are implemented where. For most purposes it isn't actually important what is implemented in the kernel and what is not. What is important is the layering of guarantees and dependencies.

- In many places, the design relies on dependencies between *instances* of servers rather than dependencies between servers. You really cannot just look at the programs to understand who depends on what. You need to look at the *processes*.
- When drivers were moved out of the kernel in later versions of EROS, the clear layering dependencies between kernel and user code became harder to see.

All of this is sufficiently unlike other systems to make getting your head around it a bit tricky.

<sup>†</sup> Copyright © 2007, Jonathan S. Shapiro. Verbatim copies of this document may be duplicated or distributed in print or electronic form for non-commercial purposes.

## 1.2 A Word on Terms

In this note, I will use the following terms:

- A **server** is a process that implements one or more objects.

Note that servers may be (and often are) multiply instantiated! There may be many “directory servers,” each implementing “directories.”

By special dispensation, the nucleus (the portion of the system that executes in CPU supervisor mode) is also a “server.” From the application developer’s perspective, however, the answer to the question “is this implemented in the kernel?” is usually: “none of your business.”

- The term “object” is more problematic, because it doesn’t have a common definition across communities in the world of computer science. In this note, an **object** is a logically coherent thing that is implemented by a server. An object is named by one or more capabilities.
- An **interface** is a collection of methods and an intended “meaning” for those methods in terms of their effect on one or more objects. In this paper, the term “interface” is used as a *type*, as in: “capability X implements the Y interface.”
- A **capability** is a nucleus-protected descriptor that names a server process and contains a unique ID whose interpretation is known only to the server.

In practice, the unique ID is generally used to identify an (object, interface) pair that is implemented by that server. So, we have the statement:

*A directory capability names a directory object within a directory server and implements the directory interface.*

## 2 Comments on Confusions

Before I get into the “meat” of the discussion, let me comment briefly on some of the sources of confusion.

### 2.1 Points of View

Both KeyKOS and EROS are object-capability systems. Informally this means that every operation in the system is performed by invoking some capability that names some object. For the most part, “object is as object does.” If I build a process P that implements a capability P.entry

whose behavior simulates the behavior of some other capability C (for example, if P simply forwards requests on P.cap to C and forwards replies back to the caller), the caller generally cannot tell. There are operations in the system that could disclose this simulation<sup>1</sup> but the client generally won’t know unless they go a bit out of their way to find out. In the eyes of most programmers, there is a funny conceptual consequence to this.

- When you look at the system from the kernel perspective, the separation between kernel and application is very clear.
- When you look at the system from the user-mode perspective, the system is simply an arrangement of objects. Some of these objects happen to be implemented by the kernel, but this is not “interesting” from the user-mode point of view. The guarantees of the system (if any) are a necessary consequence of three things:
  1. The individual behavior contracts of the object servers,
  2. The initial arrangement of object relationships (who holds capabilities to what), and
  3. The feasible evolution(s) of the system state proceeding from those operations that are authorized by the initial arrangement of objects and capabilities.

I suspect that this may be part of why the line between kernel and application in the system structure becomes blurry, so I wanted to point it out. The problem with Virtualization is sometimes a curious thing.

### 2.2 KeyKOS vs. EROS Layering

In KeyKOS and early EROS versions, drivers and the object store were implemented in the kernel directly. In these systems, the following statements were literally accurate:

1. The kernel implements a strong partition between capabilities and data. There is no interface provided by the kernel that permits data to be interpreted as a capability. There *\*is\** an interface that discloses the bit-level representation of capabilities, but the capability that provides this interface is closely held by a very small number of applications.
2. Layering is strict. The kernel does not rely on the correct behavior of ANY user-level code as a precondition for kernel correctness.

<sup>1</sup> In other discussions, the word “virtualization” has been used for this, but I am avoiding this term intentionally.

So for example, when I have stated elsewhere that “storage allocated by a space bank is exclusively held at the time of allocation” this is not a property enforced by the kernel. It is a consequence of the behavior of the space bank implementation. The space bank in turn relies on the kernel to faithfully implement its specification, but that is all.

The kernel behavioral specification would not be violated if the space bank did not obey this rule. MANY parts of the larger system specification would be violated.

In later EROS versions, drivers were moved into user-mode code, so we have a system structure that looks something like this:

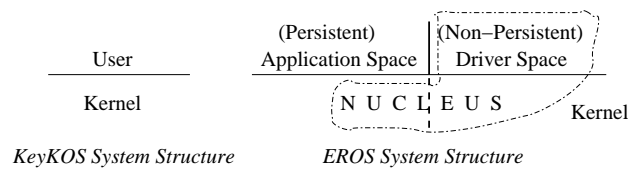


Figure 1: KeyKOS vs. EROS system organizations (kernel’s eye view).

This system structure has four properties:

1. All guarantees made by the kernel are contingent on the correct behavior of (almost) everything in driver space. That is: driver space and kernel space taken together constitute a single trust domain and reliance domain.

Specifically:

- (a) The object store implementation lives in driver space. The kernel will both import and export the raw bits of capabilities to the object store implementation, the object store in turn discloses them to the disk driver, and *\*all\** of these rely on the driver-side space bank.
  - (b) The kernel necessarily relies on the correct behavior of anything that can do physical DMA.
2. No capabilities are allowed to cross the boundary between application and driver space, because this would lead to persistence problems. The kernel implements some rendezvous objects to support communication across the boundary.
  3. While driver space is fully trusted, it is still structured using many of the same ideas and tools as application space. This is done for engineering and testability reasons, and also for fault isolation.
  4. Driver space is never persistent. It is possible to imagine a non-persistent application space (e.g. in

an embedded device), and in this case the dividing line between the two can be softer, but it is still conceptually useful to imagine a hard dividing line.

In hindsight, I really should have done a term rotation when we made this change, and I think that I will do so in Coyotos. When this restructuring occurred, I should have declared the following definitions of terms:

- The **nucleus** is the portion of the system that runs in CPU supervisor mode.
- The **kernel** is the combination of the nucleus and the “driver space” portion of the system (which happens to run in CPU user mode).
- When we refer to **user code** in the context of kernel/user discussions, we are referring to code that executes in application space. This code invariably executes in CPU user mode.

The problem here is the ambiguity of the term “user” (it refers to both CPU mode and kernel vs. application mode). By doing this term rotation I could have preserved all of the old invariant statements. Unfortunately, some of the “user code” is part of the overall system TCB, so calling it “application code” isn’t quite right. It’s a “kernel author perspective” vs. “normal developer perspective” kind of problem.

In the discussion that follows, I will use these terms as I have defined them above. When I use the term **user code**, “code running in application space”. When I use the term **kernel code**, what I mean is “the combination of code in the nucleus and driver space”. Unfortunately, I did *not* do this term rotation in the EROS papers, and in hindsight I really regret that.

From this point down I will not speak about driver space any more. Only about kernel guarantees and user guarantees. I will probably have to do a separate note concerning user guarantees.

## I The Kernel

### 3 Properties of the Kernel

The main responsibilities of the kernel are the implementation of a small number of primitive objects and the assurance that the current system state is both well-typed and consistent. It is *not* the kernel’s responsibility to ensure that the state is *safe*.

A system state is **well-typed** if all capability slots hold capabilities, all data slots hold data. It is the responsibility

of the kernel to ensure that any operation performed on a well-typed system state has well-defined results, and that one property of these results is that they constitute a well-typed state. EROS is quite permissive in this regard. More on this below.

A system state is **consistent** if it can be reached by performing a sequence of legal system operations to some initial state that was safe. By “legal,” I mean that the preconditions (permissions, validity) of each operation were tested and passed before the operation was performed. That is: a system state is consistent if the current system state is a result of applying the system operational semantics inductively. One way to ensure consistency is to ensure that all operations that can actually be articulated within the system have well defined results (possibly exceptions) under all possible well-typed conditions. This is how EROS does it.

If we know that the kernel ensures consistent states, we will later be able to choose our initial states to be **safe**. A “safe” state is one in which some security policy is achieved. With luck, it will turn out that any consistent state that is reachable from an initial safe state is safe. Sam Weber and I managed to verify this for the “confinement property” in EROS. A confinement mechanism provides a generalized building block for higher-level security policies, which is why we think it is significant. No other system has been able to verify that confinement is enforceable (in fact, quite the opposite, which is depressing). The KeyKOS/EROS confinement mechanism also ensures a more subtle property that I call “encapsulation” that supports certain kinds of integrity properties.

The distinction between “consistent” and “safe” is that the definition of “safe” is a matter of policy. The idea is for the kernel to implement a machine that supports the definition and enforcement of policy *by user code* without restricting what those policies might be. This is widely believed to be difficult or impossible. Our verification results and the Rajunas MLS design for KeyKOS [3] suggest otherwise.

### 3.1 Kernel-Implemented Objects

The nucleus of EROS implements (acts as a server for) the following objects:

**Miscellaneous Capabilities** There is a small set of miscellaneous capabilities. See the Kernel Objects section of the EROS Object Reference manual.

Of these, the ones that are mainly interesting to us are ProcessTool, Discrim, and Range. Also, the Range capability is used by space banks to allocate and destroy objects.

When considering the overall system view, it is sometimes

useful to know that the Range capability are held only by instances of SpaceBank<sup>2</sup> and the ProcessTool capability is only held by instances of ProcessCreator. These statements of exclusive possession are statements about the capability relationships that are maintained within the EROS *system*. The kernel does not rely on these statements in any way. On the other hand, if you are trying to understand how security is bootstrapped, it is useful to know that you do not need to consider the impact of arbitrary processes wielding these capabilities in arbitrary ways (*especially* the Range capability).

**Number Capabilities** A number capability is a 96 bit value. In EROS, these are used to store process register values and a few “option bits” used in the memory subsystem. Number capabilities are not used in Coyotos.

**Pages, Nodes** The primitive object types *page* and *node* and all instances of these objects. As part of this implementation, the kernel ensures that:

- The set of system pages is a disjoint set.
- The set of nodes is a disjoint set.
- A partition is enforced between pages and nodes. That is: the kernel enforces the disjointness of these types.
- The type disjointness extends naturally to capabilities and data, because capabilities may only be stored in nodes, while data can only be stored in pages.

Some versions of EROS talked about “capability pages.” These should be understood as “large nodes whose size happens to conveniently match the hardware-defined data page size.” Capability pages are nodes.

In Coyotos there are objects that store both capabilities and data in a single object, but these objects have clearly identified “slots” and the slots are statically typed. The data/capability partition is preserved, but at the field granularity rather than the object granularity.

**Processes** It is somewhat confusing whether a Process is an object or not. While there is a clear-cut process *interface*, the actual state of a process is just an assembly of Nodes. A process capability can therefore be viewed as naming a different interface on the node object or as naming an object in its own right.

<sup>2</sup> In some KeyKOS implementations, database subsystems might also hold them in order to reduce metadata overhead.

In KeyKOS there were Range capabilities that could allocate/destroy only a contiguous subrange of objects, and a capability Super-Range that could allocate/destroy *any* capability. Subrange capabilities were dropped in EROS.

In practice, the latter convention is assumed *unless* we are discussing the implementation of the ProcessCreator (a non-kernel server, see below). This works in practice because the process creator assembles the nodes into the proper “shape” and then returns a process capability to them. Since the individual node capabilities are never disclosed, the interface vs. object confusion is rendered irrelevant in practical terms *from the application point of view*.

Having said this, *the kernel does not assume that processes are fabricated by instances of ProcessCreator*. The kernel requires (and tests) a set of well-formedness constraints concerning the arrangement of nodes before it will agree to interpret those nodes as a process. If these constraints are not observed, there is a well-defined failure behavior. In fact, the kernel will behave correctly and properly if a node capability and a process capability to the same node exist simultaneously and are invoked in interleaved fashion.

In practical terms, this defensiveness on the part of the kernel is almost never observed, because the most convenient way to fabricate a process is to use a handy ProcessCreator. Still, when you are trying to understand who makes what guarantees and who makes what assumptions, In Coyotos, Processes are no longer composed of Nodes, so the interface/object confusion is eliminated.

**AddressSpace** An address space capability is really a node capability in disguise. The primary difference is that you use a Node capability to fetch from slot *i* of a Node, but you cannot use the AddressSpace capability to do this. In fact, a Node capability is legal anywhere that an AddressSpace is legal. Given this, it’s conceptually safe to view AddressSpace as just a second interface on a Node, in much the way that read-only and read-write page capabilities are two interfaces on the same page object.

An EROS address space is just a tree-structured arrangement of Nodes. As addresses are referenced by processes, the kernel attempts to translate them by traversing the tree in the obvious way (fragmenting the address to provide indexes into each level of the tree structure). An address is translatable if (a) a page capability is found before 20 nodes are traversed [this ensures bounded time operation], (b) no cycles are found during path traversal, and (c) all capabilities traversed are of “sensible type (page capabilities, node capabilities, or address space capabilities).

I’m ignoring a bunch of complications from the way that address spaces are encoded in unusual cases. The description above is conceptually sound, but a hundred details or so are omitted here for brevity.

Nodes were replaced in Coyotos by GPT objects. More flexible, more efficient, simpler encoding and traversal, but basically the same idea in the same conceptual part of

the nucleus design.

**Entry Capabilities** Start and resume capabilities (collectively “entry capabilities”) are used to send messages to servers via IPC. Their behavior within the kernel is in many respects exceptional.

## 3.2 Properties Maintained by the Kernel

The following properties are true of all invocations on kernel capabilities, except as noted. I am ignoring discussion of object paging here, which is completely transparent to applications.

**Constant-Bounded Latency** All kernel capability invocations complete (successfully or by error) in time bounded by a small constant. This implies that the kernel is deadlock free. Application deadlock is not precluded. It also implies that the kernel must operate in bounded resource. Memory requirements for every operation are bounded. There is a constant upper bound on maximum number of primitive objects touched per operation. Unbounded recursion is not permitted within the kernel.

Exception: the kernel wait object (which is how processes voluntarily sleep for some duration) does not have a kernel-imposed latency bound from an externally observable perspective. The latency from invocation to yield *does* satisfy the bound requirement.

**Atomic Units of Operation** Except for “entry capabilities” (start, resume), an invocation on a kernel capability will only modify the object named by that capability, and will be performed atomically (i.e. as a single, indivisible operation).

Exceptions:

- Allowance must be made for memory indirect invocations. The capability invoked need not appear directly in the invoking process’s state. In some cases may be obtained by traversing an address space.
- In the case of a process capability invocation, the “object” is the collection of Nodes that implement the process state.

**Complete Specification** Every expressible capability invocation has a well-defined result, even if the object named by the capability is not well-formed.

For example: on IA/32 a process is a particular arrangement of multiple Nodes. From the kernel perspective it is possible that a process will be **malformed**, meaning that (a) one or more Nodes is missing or revoked, or (b) certain slots that are required to hold capabilities of particular types [e.g. schedule slot must hold a valid schedule capability] do not. In all of these cases, the invocation of the

process capability has well-defined results (generally: an exception message).

**IPC Quasi-Atomicity** Invocations of “entry capabilities” may modify the destination process state and the receive buffer area(s) designated by the receiving process. The *data* transfer portion of the operation is performed destructively and optimistically. It will not be “undone” if there is a late page fault that cause the operation to be uncompleted. The *capability* transfer phase is done after the data phase completes as an atomic unit of operation.

The capability phase atomicity guarantee is required to ensure that every system state is consistent. It turns out that data updates have no semantics for security policy purposes (or at least, not beyond the permissions check). This is convenient, because it means that we can cheat about atomicity on the data transfer without violating the system safety induction. While system consistency *does* depend on integrity of data updates, it appears to suffice that an incoming receive buffer’s state is undefined and subject to change between the initiation of receive and its completion.

**Interface Simplicity** No kernel-implemented operation returns a data string. This facilitates atomicity in the face of multiprocessor implementations, gathering all of the really hard cases into the IPC path where such things seem to like to accumulate. As we all know, researchers generally build microkernels so that they have an excuse to play with IPC...

**Non-Bypassable Access Checks** Every kernel operation enforces the applicable permissions of the invoked capability. You cannot write through a read-only capability.

**Faithfulness to Formal Model** Every operation that is executed is a capability invocation, without *any* exception. It’s a definite stretch, but this is even true conceptually for individual instruction execution at the CPU level.

This allows us to model that the entire behavior of a machine running the EROS kernel onto our formal capability access model. The formal model provides continuous simultaneous descriptive coverage at the machine instruction, kernel invocation, and application execution levels of abstraction.

**Stateless Kernel** To first and second order approximation, all non-page, non-node state in the kernel is merely a cache, and can be discarded (possibly after write-back) and/or recreated at will. An implication of this is that no process ever blocks on a kernel stall queue with any retained kernel state. A blocked process can be almost completely paged out. The *only* part of a blocked process that must be retained in memory is a six word structure that contains the stall queue link pointers and a capability that names the blocked process.

From the “outside the kernel” perspective, a process that is blocked on a kernel stall queue appears to be in the “running” state. It just doesn’t make a hell of a lot of progress until it unblocks.

Unfortunately the “pure cache” claim isn’t perfectly true, because the list of currently running processes at the time of checkpoint doesn’t quite satisfy this statement. As far as I know that is the only exception, and it is handled as a (regrettable) special case.

### 3.3 Things to Notice

Note in the preceding subsections there is *no* higher-level policy implemented within the kernel. In KeyKOS, the object paging policy was implemented by the kernel, but it was efficient enough that nobody cared. In EROS, the object eviction (i.e. aging) policy is similarly implemented in the kernel, but the actual paging is handled by driver code.

Primitive resource multiplexing (which includes object aging and CPU scheduling) is very difficult to entirely remove from the kernel. Exokernel, in my opinion, is the most promising attempt, but the effort did not succeed in practical terms.

## II Fundamental Servers

At this point the discussion shifts to how the EROS *system* is structured, and I will abandon the kernel point of view.

The fundamental servers are the ones that *everything* else relies on. The services they provide are very low level: storage allocation (SpaceBank), process construction, identification, and destruction (ProcessCreator), process assembly and confinement (Constructor).

Well, not quite. The *real* fundamental servers are actually PrimeBank, MetaConstructor, and ProcessCreatorCreator (commonly known as PCC because ProcessCreatorCreator is unpronounceable), but I need to explain how the type system works before I can explain that. PrimeBank is the root of the SpaceBank hierarchy. MetaConstructor is a distinguished Constructor instance that knows how to fabricate new Constructor instances (saying “constructor of constructors” becomes awkward quickly). PCC is a distinguished Constructor that knows how to fabricate ProcessCreator instances.

All fundamental servers are **primordial**, meaning that they exist in the initial system image as a consequence of hand construction (via a tool). In order to have the object type system work nicely, there is one capability in the initial system image that you couldn’t get by actually

running the fundamental processes. This “Gordian Resolution” in the system is intentional. Bear with me.

## 4 Primitive Process Identification: Branding

Before we actually get in to the primordial services, I need to digress briefly to explain the idea of a **brand** and the mechanism by which processes are typed.

The kernel implements a capability called `ProcessTool` that has three functions:

1. `MakeProcessKey(NodeCap, C) => ProcessCap` Given a `Node` capability and a second capability `C`, fabricate a `Process` capability to that node, install the capability `C` into the **brand slot** of the `Process` (which is a slot in the `Node`), and return the `Process` capability to the caller.

There is no operation on a `Process` capability that reveals or modifies the brand slot of the process.

This is the bottom-level operation by which `Nodes` come to have “process nature.”

2. `Identify(ProcessCap|EntryCap, C) => Bool` Disclose whether the provided capability `C` matches the brand slot capability of the process named by the provided process or entry capability.

3. `CanOpener(ProcessCap|EntryCap, C) => NodeCap` If the `Identify()` check passes, return a `Node` capability to the process named by the provided process or entry capability.

This is used by `ProcessCreator` instances during process deconstruction. Note that the existence of this `Node` capability does *not* alter the behavior of the process. If `ProcessCreator` goes to destroy these `Nodes` using the wrong space bank, the intended victim process will continue unmolested.

The branding mechanism is puzzling until you see how it is used. Imagine for a minute that you (a process) can come up with some capability — any capability — that nobody else has. You can use this capability as a label for the processes you create, and later you can check if a `Process` is one of yours. This can be used to bootstrap a variety of process instance authentication mechanisms, most notably `ProcessCreator.hasInstance(someProc)`.

The identification mechanism that is ultimately provided by `ProcessTool` sits at the core of `EROS` generalized confinement mechanism. It allows confined subsystems to be defined recursively.

By the way, the usual choice for a “secret” capability is for the process wielding `ProcessTool` to use a distinguished entry capability to itself that is not otherwise handed out to anyone.

## 5 Primordial Servers, Take I

I need to describe the primordial servers in two passes. The first pass describes what they *do*. The second pass will address some reductio issues in the design.

### 5.1 PrimeBank and other Space Banks

Space banks exist in a logical tree that is rooted at the `PrimeBank`. The `PrimeBank` is the ultimate source of *all* `Page` and `Node` allocations. In practice, all space banks are actually implemented by a single server, commonly referred to as “the space bank” for the purpose of confusing new developers.

A space bank has five “interesting” operations (some with several variants):

1. Allocate a page or node. The space bank guarantees that the resulting capability is the only valid capability to that page or node, and will not subsequently hand out another capability to that object to anyone.

An allocation may fail due to limits (quotas). Logically, the allocation request proceeds recursively upwards in the space bank hierarchy, each bank getting the object from its parent until we reach the `PrimeBank` that actually does the work. In practice this is handled by simply decrementing the limits (if any) of all parent banks.

2. Deallocate a page or node, causing all outstanding capabilities to become invalid. Fails if object not allocated by this bank.

As a matter of implementation, the page or node is getting reused and the deallocation operation zeroes the object. A version number is maintained to ensure that no version of a page is given out twice. From a *logical* perspective, pages and nodes are not reused.

If a deallocated `Node` is occupied by a running thread of control (recall that processes are implemented by `Nodes`), the thread of control ceases to exist.

If a deallocated `Node` contains a valid resume capability, that resume capability is invoked with a `ProcessDied` exception message (not implemented). This provides a watchdog mechanism of sorts.

3. Destroy bank, causing all objects allocated from this bank *and its descendant banks* (including the descendant banks themselves) to be deallocated. A variant permits immediate child banks to be reparented to their grandparent rather than being recursively destroyed.

4. Create child space bank.

This is generally done immediately before each process fabrication, so that every process has a dominating space bank that can be destroyed as an extreme recovery measure.

5. Identify bank. One space bank can identify another. This is used to determine whether a capability you are handed actually points to an “official” space bank. You want to know this because otherwise you don’t have any guarantee that the “exclusive allocation” guarantee is actually being met.

There is a widely available bank `SpaceBankIdentifier`. This bank is a descendant of `PrimeBank` with a zero allocation limit and `NoDestroy` permission restrictions. The main use of this capability is that you can give it out to anybody so that they can identify “official” space banks. In `EROS`, the Constructor uses this capability to implement its check of `SpaceBank` authenticity.

Note that `PrimeBank` has no fault handler. It must extend its heap by hand using primitive `Node` and `Page` operations, and it must ensure that all necessary regions of the `PrimeBank` heap are backed before allocation. There is also no exception handler for `PrimeBank`.<sup>3</sup> Needless to say, bugs in the `PrimeBank` are a true joy to find and repair.

## 5.2 Process Creators

A `ProcessCreator` implements three operations:

1. `constructProcess(SpaceBankCap) => ProcessCap`. Given a space bank capability, allocate the necessary nodes, arrange them as a process, and invoke the kernel `ProcessTool` to obtain a process capability to the “root” `Node` for the process. Install a **brand** capability that is known only to this `ProcessCreator` instance for use in later recognition. Return the process capability.

<sup>3</sup> We have joked that there should be a primordial `SystemDeathNotice` fault handler, but it isn’t clear who it would talk to, because the `tty` subsystem might need to allocate memory to accept text. Realistically, the only thing (and the right thing) for `DeathNotice` to do is invoke the kernel hard reboot operation.

2. `identifyInstance(ProcessCap|EntryCap) => Boolean`. Invoke the kernel `ProcessTool` to determine whether the brand of the passed process capability matches the brand known to the `ProcessCreator` instance. If the answer is `true`, this means that this process was constructed by this `ProcessCreator` instance.

If used correctly, this operation provides an unforgeable implementation of `isTypeOf` for process capabilities.

3. `destroyProcess(SpaceBankCap, ProcessCap)` Check if the passed process capability satisfies `identifyInstance()`. If so, attempt to deallocate it by selling the constituent nodes back to the space bank named by `SpaceBankCap`, which will fail if this space bank did not allocate those nodes, having no effect on the process.

A variant is `destroyCallerAndReturn`, which tears down the calling process and returns a result code to a third party. This is used to let an exiting process to return an operation result to its last caller.

Every `ProcessCreator` instance implements exactly one `ProcessCreator` object.

## 5.3 Constructors

Like `ProcessCreator`, each `Constructor` is a single-object server. Each `Constructor` instance has a privately held `ProcessCreator` instance. On request, a `Constructor` uses its `ProcessCreator` to fabricate a new process, installs the **initial capabilities** (supplied by the process that created the constructor) into that process, and sets the process running. The end effect of this is that a given `Constructor` knows how to create instances of a particular program. Thus, we talk about “the directory constructor,” which makes instances of directories.

The main operations of the constructor are:

- `create(SpaceBankCap, ScheduleCap) => EntryCap`. Creates a new process from storage supplied by the caller. There is some internal trickery by which the return is actually performed by the newly fabricated process.
- `checkConfinement() => Boolean`. Reports whether the program that this `Constructor` creates is initially confined.
- `identifyYield(EntryCap) => Boolean`. Reports whether a given entry capability names a process created by this `Constructor`. This is

implemented internally by calling the private `ProcessCreator`.

## 5.4 The Reductio Failure

This whole arrangement is reasonably tidy. There is one space bank server, which is hand-constructed during initial system image fabrication. All other processes are constructed by `ProcessCreators` and initialized by `Constructors`.

But where do `Constructors` and `ProcessCreator` instances come from? It would be elegant to reconcile this, mainly because we would like these to fit into the `hasInstance()` framework. The problem is that a `ProcessCreator` cannot be fabricated by a `Constructor`, because the `Constructor` must first have a `ProcessCreator`.

Let me say here before we go on that the design of EROS/KeyKOS in this place was unnecessarily complicated. The `ProcessCreator` function probably should have been implemented in either the `Constructor` or the `SpaceBank`. In Coyotos, we plan to put it in the `SpaceBank` and get rid of `ProcessCreator`.

## 6 Primordial Servers Reloaded

**MetaConstructor** The simple part of the reconciliation is the `Constructor of Constructors`. Given that we can put `Constructor` instances into the initial system image, it isn't a big stretch to put a primordial `Constructor` instance in there as well. We arrange by hand for all of these `Constructors` to look like they were built by the "Constructor `ProcessCreator` instance." We then set up a distinguished primordial `Constructor` instance that knows how to fabricate new `Constructor` instances. This is the **MetaConstructor**.

As it turns out, the entire `Constructor` program can run in constant memory. We arrange things so that `MetaConstructor` allocates its working memory from `PrimeBank` at startup time. So far so good. Now we need to come up with the `ProcessCreator` that `MetaConstructor` uses to build new `Constructors`. That turns out to be the `Constructor ProcessCreator` instance.

Okay, so what about that? Simple. The `Constructor ProcessCreator` instance is *also* built by hand.

**PCC** Which brings us down to one last problem: what about the `Constructor of ProcessCreator` instances, also known as `PCC`?

At this point the *reductio* breaks down and we resort to blasting tools. There is a single program `PCC` that knows how to build `ProcessCreator` instances. This program im-

plements the `Constructor interface`, but it does not run a copy of the `constructor code`. In KeyKOS, `MetaConstructor` does *not* identify `PCC` as a `Constructor` instance, so there are three irreducible objects (`PrimeBank`, `PCC`, `MetaConstructor`).

In EROS, `PCC` is branded as a `Constructor` and will be identified by `MetaConstructor` as a `Constructor` instance. This is accomplished by reaching back and hand-installing the `Constructor` brand into the appropriate capability slot of the `PCC` process.

*A figure is needed here urgently.*

**Other Tidbits** A noteworthy property of `PrimeBank`, `PCC`, and `MetaConstructor` is that they must not *ever* take exceptions. The system may limp along a bit after one of these processes halts, but the system really just isn't going to make much more progress at that point. An open issue is how to design appropriate watchdogs for these processes.

While page fault handlers (in particular the copy on write page fault handler) are very widely used, they are *not* considered primordial and the system security policy enforcement mechanism nominally does not rely on them to honor information flow restrictions. That said, it is hard to imagine a comprehensible system in which the commonly used page fault handling mechanisms could be substantially misbehaved.

## III Non-Foundation Servers

### References

- [1] Norman Hardy. "The KeyKOS Architecture." *Operating Systems Review*, 19(4), October 1985, pp. 8–25.
- [2] 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.
- [3] S. A. Rajunas. *The KeyKOS/KeySAFE System Design*. Key Logic Technical Report SEC009-01. March 1989. Key Logic, Inc.