State Caching in the EROS Kernel
Implementing Efficient Orthogonal Persistence in a Pure Capability System

Jonathan S. Shapiro
David J. Farber
Jonathan M. Smith
University of Pennsylvania

2 September, 1996

Abstract

EROS, the Extremely Reliable Operating System, addresses the issues of reliability and security by combining two ideas from earlier systems: capabilities and a persistent single-level store. Capabilities unify object naming with access control. Persistence extends this naming and access control uniformly across the memory hierarchy; main memory is viewed simply as a cache of the single-level store. The combination simplifies application design, allows programs to observe the “principle of least privilege,” and enables active objects to be constructed securely.

Prior software capability implementations have suffered from poor performance. In EROS, caching techniques are used to implement authority checks efficiently and to preserve the state of active processes in a form optimized for the demands of the machine. The resulting system provides performance competitive with conventional designs. This paper describes the EROS object model and the structures used to efficiently map this model onto one hardware implementation: the Intel 80×86 processor architecture.

1 Introduction

EROS, the Extremely Reliable Operating System, provides an environment for long-lived persistent application systems. The motivation for this effort is to facilitate research in user environments, reliable application design, scheduling, security, and (in the future) recoverable distribution in such systems. A primary design objective is to achieve a software mean time between failures (MTBF) measured in years.

Two essential differences between EROS and conventional systems are the use of persistence and capabilities. Persistence simplifies the construction of active objects, which have greater expressive power than the passive objects of conventional systems. A consistent snapshot of all system state, including processes, is periodically written to the disk by a lightweight, fault-tolerant checkpointing mechanism. This mechanism allows EROS to recover after transient failures in less than one minute, having lost a bounded amount of work. In addition, it dramatically improves the efficiency of disk write traffic.

Capabilities unify object naming with access rights. While access control lists (ACLs) provide equivalent power in conventional systems, capabilities offer several conceptual advantages over access control lists:

- Their authority can be delegated.
- They support the “principle of least privilege.” Applications can be designed to hold no more authority than they require.

*This work was supported by the Hewlett-Packard Research Grants Program, the AT&T Foundation, CNRI as manager for NSF and ARPA under cooperative agreement #NCR-8919038, NSF #CDA-92-14924, and ARPA #MDA972-95-1-0013.
- They provide encapsulation: objects can undetectably act as proxies for most other objects without compromising security.

- They eliminate the need for the kernel to have any notion of user identity. A single system can therefore support multiple simultaneous administrative policies defined by mutually adversarial sources of human authority.

Prior software capability implementations have suffered from poor performance. In EROS, caching techniques are used to implement authority checks efficiently and to preserve the state of active processes in a form optimized for the demands of the machine. This design reduces kernel size and complexity, limits the scope of software errors, and facilitates their detection before committing their consequences to the disk. The resulting system provides performance competitive with conventional designs.

This paper presents the EROS object architecture, and describes how the architecture is efficiently mapped on to the Intel processor architecture.

## 2 The Object Architecture

EROS is a fine-grain capability system. All stateful abstractions are composed from only two primitive, fixed-size object types. Pages are the basic unit of user data storage, and contain an architecture-defined amount of user data. Cgroups are the basic unit of capability storage, and hold 16 capabilities. Cgroups are protected by partitioning. Only the kernel is permitted to examine or modify the content of a cgroup. As a result, applications are unable to examine or forge capabilities.

Pages and cgroups are composed to construct objects such as memory segments and domains of authority.

### 2.1 Capabilities

Objects and services in EROS are named by capabilities, which consist of a \((\text{type}, \text{object id}, \text{authority})\) tuple. The semantic interpretation of an object in a given context is determined by the type field of the capability. Placing the type indicator in the capability rather than the object allows us to take advantage of knowing the underlying object representations to support user-level manipulation of composite objects, for example in fault handlers. While it is possible to fabricate combinations of capabilities that are nonsensical, doing so does not violate the integrity of either the kernel or the application.

In addition to naming objects, capabilities can also name the program obeyed by a domain or an operating system service. They therefore subsume both system calls and interprocess communication.

The major capability types supported by the EROS kernel are shown below. Types shown in bold are objects constructed from cgroups and pages. Types shown in italic are services implemented directly by the kernel. These are shown for completeness, but are not described by this paper.

<table>
<thead>
<tr>
<th>Page</th>
<th>Cgroup</th>
<th>Number</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Start</td>
<td>Resume</td>
<td>Sense</td>
</tr>
<tr>
<td>Schedule</td>
<td>Misc</td>
<td>Device</td>
<td></td>
</tr>
</tbody>
</table>

Pages and cgroups have already been described. The corresponding capabilities exist in read-write and read-only variants. Segment capabilities can also be read-write or read-only.

A **number** capability is a self-describing object containing a 96 bit unsigned constant that can be fetched by the application. They are primarily used in domains, as we will see shortly. A number capability whose value is zero is referred to as a **null capability**.

A **sense capability** is similar to a read-only cgroup capability, except that its read-only property is “sticky.” The holder of a sense capability cannot obtain read-write access to any object that is transitively reachable through the sense capability.
2.2 Segments

Memory segments (“segments”) are the basic data organization mechanism of EROS. They are constructed by assembling cgroups into a tree whose leaves are pages (Figure 1).

The segment structure is similar to page tables in conventional memory mapping architectures, and structurally similar to the file system indirection blocks of UNIX [Tho78]. EROS address spaces are implemented as segments. Data access rights are determined by starting at the segment root and walking down the tree to the page; if any capability on the path from the root of the tree to the page is read-only, then all accesses within the corresponding subsegment are read-only, even if read-write capabilities exist within the subsegment.

As in hardware address spaces, segments may have undefined subregions. This is used to describe unmapped portions of a contiguous area or unallocated portions of a program heap.

Unlike hardware address spaces, the height of an EROS segment tree is variable. The size of an EROS segment is $16^h$ pages, where $0 \leq h \leq 30$ is the height of the tree. EROS segments therefore range in size from 1 to $2^{120}$ pages. This is adequate for most currently envisioned databases, and can readily be extended. The biased log of the segment size (BLSS) is contained in its capability. BLSS is defined as

$$\log_{10} (\text{segment size in bytes}) - 1$$

The bias comes from the fact that the value is almost always used in traversing cgroups while walking segment trees, where the index desired at a given cgroup is given by

$$\text{(segment offset} \gg \text{BLSS)}$$

2.2.1 Red Segments

The simple segment cgroups described above are known in EROS as black segments. A special sort of segment known as a red segment exists to construct more complex segment structures. Red segments provide for the identification of the segment’s fault handler, and for segments of variable size, and for “background segments.”

We mention red segments primarily to make it clear that EROS provides efficient support for copy-on-write and a mechanism for user-level handling of memory access exceptions, but will not address them further in this paper. For a more complete discussion, the reader is referred to the online EROS documentation [Sha96a, Sha96b].

---

1For this reason, following the conventions of KeyKOS [Har85], cgroups are called nodes in the bulk of the current EROS documentation. To avoid collision with the distributed systems community, we are abandoning this term in favor of “cgroup.”
2.2.2 Segmode Capabilities

Given a cgroup capability, the holder may ask for a segment capability for the same cgroup. Cgroup capabilities and segment capabilities can be used interchangeably to identify a segment. Cgroup capabilities expose the structure of the segment to the holder, while segment capabilities keep the internal structure opaque. Typically, a segment will be constructed out of cgroups using cgroup capabilities, and a red or black segment capability is then fabricated for the root cgroup and handed to the user. Similarly, a page capability may be used for sufficiently small segments. The term segmode capability is used in contexts where any of these three types of capability is acceptable. The slots of a black segment cgroup contain segmode capabilities.

2.3 Domains

In reviewing the CAL/TSS project, Butler Lampson noted that domains of authority and processes should have been unified [Lam76]. Similar conclusions are suggested by Wulf’s postmortem on HYDRA/C.mmp [Wul81]. EROS unifies domains of authority and processes into a single object known as a domain (Figure 2).

A domain is a cgroup (known as the domain root) which contains:

- A schedule capability. The EROS scheduler implements processor capacity reserves [Mer93]. Schedule capabilities convey the authority for a running domain to execute instructions under a particular scheduling reserve.
- A keeper start capability naming the domain (the keeper) responsible for handling execution faults incurred by this domain.
- A segmode capability naming the address space for the domain. The domain is said to obey the program embodied in its address space.
- A cgroup capability to the capability registers cgroup, which serves as the capability registers of the domain. The application may invoke a capability giving the index of the capability register that contains it.\(^2\)
- A cgroup capability to the general registers cgroup. The general registers cgroup contains number capabilities holding those register values for the domain that could not be squeezed into the available slots of the domain root. On the 80x86 architecture, it is required only for domains that use the floating point unit.
- A brand, which is a capability used by the fabricator of the domain to identify the domains it constructed.
- A status, which is a number capability containing the current state of the domain (running, available, or waiting) and the current fault code and fault information (if any).

Each of these capabilities resides in a designated slot within the domain root. The remaining slots of the domain root are used to hold the register values for the domain.

Figure 2: Domain with keeper

\(^2\)The capability registers cgroup may be omitted for emulation domains, since these domains cannot invoke capabilities.
Possession of a domain capability allows the holder to alter most of the slots of the domain. Domain capabilities thus convey authority over the structure of the domain. By contrast, **start capabilities** and **resume capabilities** convey the authority to invoke the program embodied in a domain. Because the program and its associated state are persistent, a start capability in effect names an active object.

In its role as a process, a domain can be in one of three states:

**Available**  An available domain can be invoked by any holder of a start capability for that domain. Domains return to the available state when their program performs a return invocation.

**Running**  Once invoked, a domain moves from the available state to the running state. A running domain is servicing an invocation. If a start capability is invoked while the domain is running, the invoker will block until the domain becomes available.

**Waiting**  A domain that has invoked a capability and is waiting for a response moves from the running state to the waiting state. It remains in this state until a resume capability to the domain is invoked.

The vast majority of domains in an EROS system are available at any given time. If a domain is not available when a start capability is invoked, the invoker will block until the domain becomes available. If a domain is not waiting when a resume capability is invoked, the invocation proceeds as though a null capability had been invoked.

### 2.4 Capability Invocation

EROS objects are manipulated by invoking capabilities. An invocation names the capability to be invoked (the index of the slot in the capability registers cgroup containing the capability), and a message to be sent to the invoked object. The message includes an operation code identifying the operation requested (in the reply, this is a return code), up to 64K of contiguous data, and four capabilities (indices into the capability registers). In addition, the invocation specifies which capability registers should be overwritten by the replying invocation, where the reply data should go, and how many bytes of the reply should be accepted. Capability register zero is “hardwired” to the null capability, and is used in place of capabilities that should not be sent or received.

There are three types of invocation: call, return, and send. A call invocation blocks for a response. It places the invoker in the waiting state and the recipient in the running state. A return invocation is the inverse of the call invocation. The invoker is placed in the available state and the recipient in the running state. The send invocation sends a message without waiting for a reply; both invoker and recipient end up in the running state.

An unusual characteristic of the EROS invocation mechanism is that the kernel does not maintain a call stack. Instead, every domain has an associated call count. The call operation generates a resume capability containing the current call count of the domain. When a valid resume capability is invoked, the call count is incremented. If the call count of the domain and the call count of the resume capability do not match, the resume capability acts like a null capability. Among other uses, this allows a domain to act as a single entry point for a multithreaded service, as shown in Figure 3.

![Figure 3: User level demultiplexing](image-url)
2.5 Threads

It is useful to distinguish between a running domain and the thread of control that occupies it. A thread associates a running domain with a particular processor group. Threads are the unit acted on by the EROS kernel scheduler. EROS threads are stateless, and are second class abstractions in the sense that they have no associated capability.

A thread occupies exactly one domain, and a domain occupied by a thread is in the running state. When a domain performs a call or return invocation, its thread migrates to the new domain. If the two domains operate under the same schedule capability this action does not require a rescheduling call, which significantly reduces the complexity of the IPC path. The result is similar to the migrating threads of Mach 4.0 [For93] or the shuttles of the Spring [Ham95] system. Once we have a chance to tune the paths, we expect to achieve context switch times comparable to with L3 [Lie93b] and Mach 4.0.

A further reason to distinguish threads and domains is that the thread embodies the smallest part of a domain that must be kept in memory when the domain is not executing instructions. Domains such as login agents and detached user interfaces spend most of their time blocked in the kernel while waiting for an event to occur. Isolating the thread structure allows these domains to be paged out.

3 Mapping to the Machine

The simple object model provided by EROS is relatively easy to reason about and easy to render persistent, but it is not intrinsically efficient. The basic abstractions provided by EROS must be mapped onto the underlying hardware representation, and consistency must be maintained between the abstractions and the resulting representation. This mapping is the key to an efficient software capability implementation. The mapping must:

- Ensure that the object named by a valid capability is in memory when needed, and that an efficient mapping exists from the capability to the in-memory object.
- Ensure that only one interpretation of a cgroup (segment, domain constituent, etc.) is required during any given invocation.
- Construct and maintain a mapping from segment cgroups to the hardware address mapping mechanism.
- Provide a representation for domain state that facilitates efficient context switching.
- Ensure that the semantics of reads and writes to cgroup slots are preserved by these optimizations.

To perform this mapping, the EROS kernel implements structures that serve as architecture-specific caches of state whose definition is in cgroups.

3.1 Preparing Capabilities

Capabilities have two forms: unprepared (on-disk) and prepared. In its unprepared form (Figure 4), a capability contains the object identifier and version number of the object. Whenever a capability is dereferenced, it is first prepared. Preparing a capability has two effects:

<table>
<thead>
<tr>
<th>rw</th>
<th>wp</th>
<th>Type</th>
<th>SubType</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Version[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OID[47:32]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Version[47:32]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OID[31:0]</td>
</tr>
</tbody>
</table>

Figure 4: An unprepared capability
- If necessary, the object named by the capability is brought into main memory.
- The version number of the capability (or in the case of a resume capability, the call count) is compared to the version number (call count) of the object. If they do not match, the capability conveys no authority and is converted to a null capability.\(^3\)

Once prepared, the capability points to an object table entry, which in turn points to the object (Figure 5). Every capability has a \textit{prepared} bit (P) that indicates the current form of the capability. The object named by a prepared capability is guaranteed to remain in memory for the duration of the current invocation.

Before being written to disk, a capability is first converted back to the unprepared format. This ensures that kernel memory pointers are never written to the disk. All existing capabilities to an object can be invalidated by incrementing the object’s version number. Doing so guarantees that all access to the corresponding cgroup or page has been rescinded.

### 3.2 Preparing Cgroups

A given cgroup can be interpreted in several ways:

1. As a raw cgroup,
2. As a domain root,
3. As the general registers of a domain,
4. As the capability registers of a domain,
5. As a node in a segment tree.

Only one of these interpretations is valid during any single capability invocation.\(^4\) A cgroup that is interpreted as a domain root will not be interpreted as a segment during the same invocation.

As with capabilities, a cgroup must be prepared before it can be referenced in a particular context. If a hardware address mapping table depends on the capability values of a given cgroup, that cgroup must be prepared as a segment cgroup. If a context structure (Section 3.3) depends on the capability values of a given cgroup, that cgroup must be prepared as a domain root, general registers, or capability registers cgroup. The current interpretation of a cgroup is cached in the in-memory cgroup data structure, and serves to identify how the state of the cgroup’s capabilities may be cached.

Every capability slot in a cgroup includes \textit{write hazard} and \textit{read hazard} bits (The “w” and “r” fields in figures 4 and 5). If a cgroup is prepared, some or all of its capabilities may be hazarded. A write hazard indicates that some machine-specific data structure depends on the current value of the capability, and must be invalidated before the capability slot can

\(^3\)Incrementing the version number of the object invalidates outstanding capabilities, allowing objects to be securely reused.

\(^4\)It is possible to construct a domain whose address space segment whose cgroup is in fact the domain root. Such a domain \textit{is malformed}, and will not execute instructions.
be written. A read hazard indicates that the current value of the capability is not up to date, and must be written back from the machine-specific data structure before the capability can be read. Read hazards typically apply to capabilities containing domain register values. All read hazards are also write hazards.

Like capabilities, cgroups are always written to the disk in unprepared form. Unlike capabilities, the interpretation of a cgroup may change over time. Before preparing a cgroup in a new way, the cgroup is first deprepared to its raw form. A side effect of depreparing the cgroup is to invalidate or flush all cached copies of the cgroup’s state. This guarantees that:

- No stale cache state remains when the cgroup is written to the disk.
- A cgroup has exactly one semantic interpretation at a time.
- Malformed objects, such as a domain root whose address space capability points to the domain root cgroup, are detected.

### 3.3 The Context Cache

In conventional operating systems, processor state is saved to an interrupt stack. If the current thread of control is preempted, this state is then transferred to process structure whose layout is carefully tuned to the processor architecture.

The EROS domain layout is chosen for the convenience of the abstract machine. Number capabilities provide a space-efficient way to store register values in a cgroup, but this representation is not efficient for loading and saving registers during a context switch. Instead, the register values of a domain are loaded into an architecture specific context structure before the process is loaded onto the hardware. The cgroups of the associated domain are prepared as domain root, general registers, and capability registers, and the hazard bits of the associated slots are set to indicate that the values of the capabilities that make up the domain are cached.

When necessary, information is selectively flushed from the context structure back to the domain cgroups to clear these hazards. The context maintains a separate hazard mask indicating which parts of the context need to be reloaded before the context is ready to run.

It is rare for an invocation to require that the register values be flushed back to the domain. Once the register values are loaded into a context structure it is easier to fetch them from the context structure than from the domain. EROS maintains a cache of the context structures for recently activated domains, avoiding the overhead of unload and reload. The state of the active domains therefore remains in machine-specific form most of the time.

The 80x86 processor automatically writes process state to the supervisor stack when an interrupt or system call occurs. To take advantage of this, the EROS kernel arranges for the supervisor stack pointer to point to the top of the context cache entry for the active context. The context structure is laid out in such a way that the values written to the supervisor stack by the processor are written directly to the appropriate locations in the context structure. Once the registers have been saved, the supervisor stack pointer is reloaded to point to an interrupt stack. Similar techniques are used in L3 [Lie93a] and Mach 4.0 [For93] to avoid copying state from the stack to the process structure.

### 3.4 Address Space Management

Address space management in the EROS kernel consists of:

- Constructing address mappings in response to page faults.
- Ensuring that any write to a segment cgroup is properly reflected in the hardware-specific data structures.
- Ensuring that mapping entries are properly invalidated when cgroups and pages are removed by aging.

Constructing the hardware mapping tables is a simple matter of walking the domain’s segment tree starting at the address space capability to determine the validity and access rights of the mapping. If no mapping is found that satisfies the access type (read or write), a segment-designated fault handling domain – the segment keeper – is invoked.
Regardless of architecture, EROS tracks page dependencies by maintaining an inverse mapping from page frames to mapping table entries. Before a page is removed the page dependency table is traversed to invalidate the dependent mapping entries.

The 80x86 implements a two-level page table with authority verification at both levels. A mapping entry (at either level) is constructed by walking a segment tree, and depends on the values of the traversed cgroup slots. If any of these slots is later modified, the corresponding mapping entries must be invalidated. To guarantee this, the address of each traversed slot is associated with the address of the generated mapping entry in a *slot dependency table*, and the slot is marked as write-hazarded. Before a write to such a slot can proceed, the hazard must be cleared by traversing the slot dependency table and invalidating all dependent mapping entries.

Segments typically range from three to five cgroups in height. If the slot dependency table is constructed in a naive fashion, every bottom-level mapping entry will therefore have between three and five dependency table entries depending on the height of the segment tree. In a tree structured mapping system, mapping entries in upper level mapping tables need their own additional dependency entries. The space overhead of this is quite large, and must be reduced.

First, we note that the dependency tables are not a *complete* mapping from slots and pages to mapping entries. It is sufficient for the dependency tables to capture the fraction of all mappings that are actually being referenced. The size of the dependency table is therefore a function of dynamic dependencies rather than static dependencies.

The 80x86 provides hierarchical authority checking. As a result, the segment slot dependencies can be built hierarchically as well (Figure 6). The reduced construction is correct because the upper level mapping entry permissions override the lower level permissions.

In addition, if we view the segment slots as projecting a shadow onto the page table entries that depend on them, it develops that the shadow of a given cgroup slot is contiguous in the mapping table. This can be leveraged by the slot dependency mechanism to transparently coalesce dependency entries into ranges. Together these optimizations substantially reduce the size of the slot dependency table.

4 Checkpoint

EROS implements persistence and exogenous fault recovery using a recoverable checkpoint mechanism. Before any object may be modified in memory, space is reserved for it in the log. On pageout, dirty objects are appended to the log. Object faults are satisfied first from the log and then from the object’s home location (Figure 7).

When a checkpoint is taken, computation is temporarily suspended and all dirty objects are marked, disabling further modification. Copy-on-write techniques enable several gigabytes to be marked in under 100 ms. Once the mark pass completes, computations is resumed and the dirty objects are asynchronously written to the log.

When the log writes have completed, a migrator is started to copy the objects back to their home locations. The check-
The checkpoint/migrate approach ensures that the system is always able to recover from the most recent successful checkpoint, even if a failure occurs while taking a checkpoint or performing a migration.

The EROS checkpoint mechanism is similar to that of KeyKOS [Lan92], but the use of a circular log makes it more adaptable to runtime load variations. In addition, a circular log structure allows the migrator to proceed incrementally [Gra93]. If there are heavy demands on memory, the migrator can move a small number of objects, write a new checkpoint directory, and update the checkpoint log header to reflect the new “most recent checkpoint” directory.

The circular log also allows us to continue reloading objects from the log until the space is reused, which improves disk arm locality. A recent examination of disk write traffic under Linux suggests that over half of the disk write traffic during a full rebuild of the EROS source tree is done to update file system metadata. Nearly all of these writes are eliminated in a recoverably persistent system. Better still, the checkpoint log allows migration to proceed in order of disk destination, which yields significant improvements in write bandwidth relative to conventional file systems.

5 Status and Conclusions

The system described in this paper is currently running, though the checkpoint mechanism is still being debugged. The design and implementation meet all of the principles set forth in the introduction. Even disk I/O delay bounds can be stated for the current implementation.

At present, the code is broken down approximately as follows:

<table>
<thead>
<tr>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14,424</td>
<td>Machine-independent C++ code</td>
</tr>
<tr>
<td>17,254</td>
<td>Machine-specific C++ code</td>
</tr>
<tr>
<td>1,696</td>
<td>Machine-specific assembler</td>
</tr>
</tbody>
</table>

The system is not yet ready for distribution, but will be made available for use by interested researchers. To join the information list, please see the EROS home page at http://www.cis.upenn.edu/~eros

Stability

After a few weeks of shakedown, the kernel has proven surprisingly robust. To stress test the early system, we implemented kernel daemons to perform constraint checking and object invalidation, and ran these at 20 ms intervals. In addition, we set the hardware clock to interrupt at approximately 0.5ms intervals. We then tried to run the system on a 20Mhz 386SX. Somewhat to our surprise, it runs, makes real progress on user domain execution, and does not miss an interrupt.

Performance

In the prototype’s untuned IPC implementation, a one-way domain-to-domain invocation takes 3520 cycles (user to user) on a 133 Mhz Pentium (26.5 µS), as reported by the hardware cycle counter. Of these, roughly 100 cycles are taken by the hardware privilege crossing mechanism. Subsequent tuning has reduced this to 312 cycles (2.34 µS). We will not reach the
lower limit of 250 cycles [Lie93a] reported by Liedtke for the 486DX50; the Pentium has a higher CPU to memory cycle ratio than the 486, and some cycles are required in our IPC path for validating the invoked capability and for the capability copy portion of the invocation.

**Correctness**

A key objective of the EROS effort is to sustain a mean time between software failures measured in years. A further objective, and in some ways a more important one, is to ensure that we never write an inconsistent checkpoint image to the disk. The worst problem in persistent systems is that errors are persistent too.

The careful tracking of dependencies and interpretation contexts allows us to ensure with high likelihood that states written to the disk are consistent. Verifiable consistency constraints can be derived from them, and are applied before the system commits a checkpoint image. If any consistency constraint is violated, the system restarts rather than committing a damaged checkpoint image.

These checks provide protection against some insidious errors, including many unreported memory errors. The state constraint tracking in effect provides a degree of redundant encoding, which makes localized memory errors in cgroups, the object table, and the in-memory object headers relatively detectable. The system even does an occasional checksum on unmodified pages.

These measures are sufficient to protect the system overall, but they do not allow us to catch memory errors in dirty user pages. We expect that these will eventually become the dominant source of data corruption on current commodity hardware, and can be prevented only with hardware support.

**Related Efforts**

EROS is closely related to KeyKOS [Har85, Bom92], a system developed by Key Logic, Inc. to support reliable time sharing services among mutually suspicious users. The EROS microkernel design borrows heavily from their experiences. The implementation is completely new, and EROS departs from the KeyKOS architecture in two ways. First, we have abandoned the KeyKOS meters notion in favor of processor reserves. Second, KeyKOS had no notion of a thread. Threads were originally introduced in the EROS kernel to support precise scheduling of drivers, and to provide a locus at which scheduling policy might be attached independent of the domain for research purposes. Schedulable kernel drivers are proving to be a significant and useful facility, but it is not clear that they warrant the overhead of the thread abstraction.

With the addition of memory reserves (pinned objects), we believe that EROS could be used in many real-time applications. The overhead of the checkpoint mechanism should be both predictable and schedulable. Persistent active objects, however, raise a number of questions concerning scheduling, particularly in the area of priority inheritance. When the client/server relationship is transitive, or (as in EROS) does not observe a strict stack-oriented call/return discipline that preserves thread identity, the priority inheritance mechanisms proposed by Tokuda et. al. [Kit93] are inadequate. Further, it is not at all apparent that priority inheritance is an appropriate model for communications between conceptually independent active agents.

**Future Directions**

The current EROS system was constructed as a baseline for a future distributed system. The single-level store and careful dependency tracking lend themselves well to Lamport’s vector time notions [Lam78] and his subsequent work on distributed snapshots [Cha85]. By combining the logging checkpoint approach with careful coloring strategies, we hope to be able to implement a recoverable distributed system with acceptable space and performance overhead.

**Acknowledgements**

Bryan Ford of the University of Utah has actively participated in the design discussions that led to this design, as have Norman Hardy, Charles Landau, and William Frantz of the Key KOS group.
Early readers caught a number of errors – some substantial – in this paper. Thanks to Mitchell P. Marcus, William Frantz, Colin McLean, Dennis Allison, and Ben Chen for their efforts in reviewing under severe time pressure.

References


