February 1996

Consistency Management in the EROS Kernel

Jonathan Shapiro
University of Pennsylvania

David J. Farber
University of Pennsylvania

Jonathan M. Smith
University of Pennsylvania, jms@cis.upenn.edu

Consistency Management in the EROS Kernel

Abstract

EROS is a persistent operating system targeted towards managing resources with great longevity. The system provides a persistent single level store supporting two fundamental object types: *nodes* and *pages*. All primary objects, including memory segments and protection domains, are constructed out of these fundamental objects, and inherit their persistence. EROS is a pure capability system: access to objects is provided exclusively through the invocation of kernel enforced, secure capabilities. This paper describes the EROS Abstract Machine and the mechanisms used to achieve efficient consistency management within the system. The implementation, including all primary objects, a low overhead checkpoint/migration subsystem, and an efficient interprocess communication mechanism, requires less than 64 Kbytes of supervisor code (prior to size tuning).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No.MS-CIS-96-06.

Consistency Management in the EROS Kernel

MS-CIS-96-06

Jonathan S. Shapiro
David J. Farber
Jonathan M. Smith

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

1996

Consistency Management in the EROS Kernel

Implementing Efficient Orthogonal Persistence in A Pure Capability System

Computer and Information Sciences Technical Report MS-CIS-96-06

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

24 February, 1996

Abstract

EROS is a persistent operating system targeted towards managing resources with great longevity. The system provides a persistent single-level store supporting two fundamental object types: *nodes* and *pages*. All primary objects, including memory segments and protection domains, are constructed out of these fundamental objects, and inherit their persistence. EROS is a pure capability system; access to objects is provided exclusively through the invocation of kernel-enforced, secure capabilities.

This paper describes the EROS Abstract Machine and the mechanisms used to achieve efficient consistency management within the system. The implementation, including all primary objects, a low overhead checkpoint/migration subsystem, and an efficient interprocess communication mechanism, requires less than 64 Kbytes of supervisor code (prior to size tuning).

Contents

[*]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.

1 Introduction

EROS, the Extremely Reliable Operating System, provides an environment for the construction of persistent application systems. The motivation for this effort is to facilitate research in user environments, reliable application design, scheduling, security, and recoverable distribution in such systems. EROS may be viewed as a single large-scale persistent application that serves multiple mutually suspicious users. A primary design objective is to achieve a software mean time between failures (MTBF) measured in years. This is achieved through a combination of careful object design, enforcement of containment, aggressive fault detection, effective fault isolation, and user-provided recovery policies.

EROS is similar to a prior system. KeyKOS [Hardy85], developed by Key Logic, Inc. to support reliable time sharing services among mutually suspicious users. Like KeyKOS, EROS implements global orthogonal persistence based on a simple fundamental object model; all system state, including processes, are checkpointed on a periodic basis. Also like KeyKOS, EROS is designed as a small microkernel with a high performance message passing subsystem [Bomberger92].

Unlike KeyKOS, EROS is designed as a distributed, real-time system. Threads in EROS are first class objects associated with a particular compute resource. KeyKOS meters have been abandoned in favor of schedule capabilities, which distribute more easily and are better suited to real-time scheduling requirements. EROS implements a distributed single level store. These changes have necessitated an entirely new implementation that departs significantly from the KeyKOS system.

This paper describes the EROS persistent system, including all of the significant primary objects. We describe the fundamental objects, the abstract machine crafted from those objects, and the mechanisms used to efficiently and consistently map these abstractions onto the underlying hardware. In addition, we describe a number of cacheing techniques used to facilitate efficient execution.

With minor differences in the sizes of a few fields, the system described here is currently running. While untuned, the techniques used are similar enough in spirit to prior implementations that we are confident they will perform well as the implementation is refined.

2 The Persistence Layer

The **primary objects** of the EROS system are *domains*, *segments*, *stall queues*, *threads*, *nodes*, and *pages*. From one perspective, these object types have co-equal status. Each is persistent, each has a well defined, kernel implemented object protocol, and each can be manipulated by any holder of an appropriate capability.

The persistence architecture is defined in terms of nodes, pages, and threads. Domains, stall queues, and segments are composed from nodes and pages. Their persistence is a consequence of their composition from these indivisible units. Where it is important to distinguish between these layers, this paper refers to nodes, pages, and threads as **fundamental objects**.

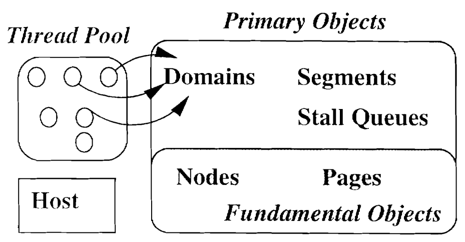

Figure 1: The Object System

A **page** is a repository of user data. It contains an architecture-defined number of bytes. A **node** is a repository of a fixed number of secure capabilities known as **keys**. In all current EROS implementations, nodes contain 16 keys. A **thread** is a stateless binding agent between a domain and a host. Threads therefore act as the locus of scheduling. Threads, nodes and pages are persistent.

All EROS objects are accessed exclusively through kernel-implemented, secure capabilities. Posession of a key for an object is a necessary *and sufficient* condition for accessing that object with the authorities conveyed by that key. A read-write page key, for example, conveys the authority to examine or modify a particular persistent page.

Every node and page has a unique (possibly duplexed) **home location**, which defines the object's unique identifier (OID). Main memory is used as a cache of the persistent store. When a key is referenced, the object named by the key is faulted into memory from the disk. Once the object is in memory, the referencing invocation proceeds on the in-memory copy. At some later time, the object will be written back to the persistent store, making the modification permanent. Efficiency is achieved by cacheing this state in a form convenient to the hardware, and ensuring that the cached images and the objects remain consistent.

Objects on the persistent store are organized in contiguously numbered clusters known as **ranges**. Every range has a type (node or page), a starting OID, and an ending OID. Once the appropriate range is located, the offset of that object within the range can be calculated by straightforward arithmetic.

2.1 The Object Cache

At startup time, the EROS kernel allocates a small number of static data structures. Machine configuration and driver initialization code then allocate whatever dedicated memory is required to support the hardware present on the machine. All remaining memory is used as an object cache for nodes and pages. On a Pentium PC with 16 Megabytes of memory, our (bloated) research kernel's object cache holds 3497 pages and 3572 nodes.

Prior to its first use, a key to a node or page contains a type, a 16-bit key data field, the object identifier, and the object's allocation count. The format of such a key is shown in Figure 2.

000 Type	**SubType**	**Key Data**
Allocation Count[31:0]		
OID[47:32]		**Alloc Count[47:32]**
OID[31:0]		

Figure 2: An unprepared key

Objects in memory are linked into a hash table on the basis of their OID. When a key to an object is first invoked, the object cache hash table is searched to determine if the object is in core. If necessary, an object fault is initiated to bring the object into the main memory cache.

Once the object is found in memory it is consulted to locate its current **object table entry**. If no object table entry exists for the object, one is allocated from the **core object table**. The object and the object table entry point to each other. When an object table entry has been located, the key is converted into its *prepared* form, which points to the object table entry. This provides efficient access to the object for future references. The prepared form of the key is shown in Figure 3.

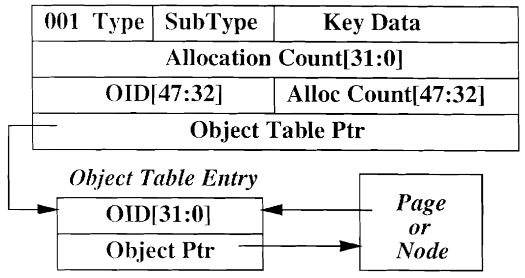

Figure 3: A prepared key

2.2 Object Versions

All pages and nodes are initially owned by a user-level domain known as the **space bank** (see below). Applications wishing to obtain storage must obtain that storage from a space bank. A space bank holds the authority to fabricate page and node keys for all of the objects in the ranges it controls. Space banks are trusted system components.

Over the course of its lifetime, a given page or node may be acquired and relinquished by many different users. Keys to this object can be copied arbitrarily by the holder, so it is necessary to have a way to ensure that a previous user cannot access the new user's data. This is accomplished by use of a 48 bit **allocation count**, which tracks the number of times that an object has been allocated.[1] Both the object and it's associated keys have an allocation count.

Whenever a prepared key is referenced, the allocation count of the key is compared against the allocation count stored in the object header.[2] If the allocation count in the key does not match the allocation count in the object, the key is invalid, and conveys no authority on the object. A key that is discovered to

[1]Given that the designed MTBF of the EROS system is measured in years, the possibility of allocation count overflow must be considered. Assuming a 32 cycle round-trip invocation time (an implausibly small number), a 48 bit counter takes 2^{53} cycles to roll over. On a machine with a femtosecond (2^{-15} sec) clock, this works out something over 8 thousand years. While its conceivable that a single system image might run that long, we feel reasonably confident that we can scavenge the persistent store every few thousand years without noticable overhead.

[2]Node object headers are kept with the object. Hardware constraints require that pages be placed in physical memory at page addresses. The object "headers" for pages are kept in a parallel data structure rather than with the page itself.

have an invalid invocation count is converted by the kernel to a *number key* containing the number 0.

When an object is sold back to a space bank, its allocation count is incremented. This is known as *rescinding* the object, and ensures that "dangling keys" to the object are rendered impotent.

The decision to accept an extra level of indirection for key validation is a deliberate engineering tradeoff. In the vast majority of key invocations, only one key needs to be validated for the invocation to proceed. Key invocations involve a privilege transition, so the additional memory reference is essentially lost in the noise on most architectures.

In the case of an interdomain call, several keys may be copied by a key invocation. Copied keys need not be validated: if the original is stale, the copy will be just as stale, and can be invalidated in a lazy fashion.

2.3 Ageing and Scavenging

Objects are placed on a "free list" by an ager, and reallocated in response to memory demands. Objects on the free list retain their content, and can be recovered if they are discovered to be in active use.

When a node is removed from memory by the ager, its contained keys are first converted to their on-disk format. If the key is prepared, its associated object table entry is consulted to obtain the low word of the key's OID, and this OID is rewritten into the key data structure.

The object named by a key may be removed from memory before the key is removed. When this occurs, no attempt is made to locate the keys that point to the corresponding object table entry. The object pointer field of the object table entry is simply set to `1`, rendering the object pointer invalid (the low bit is tested whenever a key is referenced).

As an efficiency, a new object table entry is allocated every time an in-core object is rescinded. In this event, the object pointer field is set to `3`. This allows invalid keys to be deprepared to the null key, which eliminates the need to fault in the object later to discover that the key is invalid.

Stale object table entries are recovered by the **OT Scavenger**, a low-priority background task. The OT scavenger first passes over the object table entries, using an available bit in the invalid object pointer to indicate which entries are to be cleaned. It then passes through key space, depreparing all keys that reference object table entries marked for cleaning. Finally, it frees those object table entries. Making two passes over the object table allows the OT scavenger to operate without disrupting ongoing computation.

3 The EROS Abstract Machine

EROS defines an abstract machine using nodes and pages as the basic building blocks. The abstract machine is mapped onto the underlying hardware using the mechanisms described later in this paper. With the exception of threads, all of the pieces of the abstract machine are built out of nodes and pages, and therefore inherit their persistence. Thread persistence is managed specially by the kernel.

The EROS abstract machine exposes the basic node, page and thread objects, and provides four additional abstractions: *domains*, *segments*, and *stall queues* (the last, a means for arranging efficient invocation retry, is not discussed by this paper). In addition, the abstract machine defines the mechanism by which objects are accessed: *invocation*.

3.1 Domains

Domains are assembled from nodes, one of which acts as the **domain root**.[3] A domain consists of:

- An *address space segment*, containing the program that the domain obeys and any data that the domain may construct.
- A set of *general registers*, which change as the domain executes. The register set includes all of the non-privileged registers of the underlying machine architecture.
- A set of 16 *key registers*, which identify the services that the domain can invoke.
- A priority key, defining the priority at which the domain should be scheduled.
- A *keeper*, which is a *start key* (see below) to another domain that is invoked when a domain takes an exception. The keeper is established when a domain is first created. Most domains are unable to change their keeper.

Register values in a domain are stored in *number keys*. A **number key** is a self-describing capability containing (in the current implementation) a 10

[3] In all current implementations, domains occupy three nodes: the domain root, the general registers node, and the key registers node. The general registers node holds any register values that do not fit within the domain root, and may turn out not to be required for i486 domains.

byte data string.[4] The zero number key is also known as the **null key**.

3.1.1 User-Defined Objects

A domain acts as the interpreter of the program contained in its address space, interpreting that program according to the user-mode instruction set of the underlying hardware architecture. In this sense, domains are both objects in their own right and implementors of objects. The holder of a **domain key** can fabricate a key known as a **start key**. The invoker of a start key invokes the program embodied in that domain. Once started, the program runs at the priority of its interpreting domain.

The **key data** field of a start key is set as part of its fabrication, and is passed to the recipient program by the kernel whenever that start key is invoked. Because it does not originate with the caller, the key data value is unforgeable. It can be used by the receiving program to denote multiple clients, multiple authorities, or choose among interfaces.

3.1.2 Domain States

In addition to providing services, domains provide a mechanism for synchronization. Domains are single-threaded. A domain can be occupied by at most one thread. If a client invokes a domain that is currently running, the client blocks until the domain becomes available. More precisely, a domain can be in one of three states:

Available An *available* domain is currently idle, and can be invoked by any holder of an appropriate *start key*. This is the state of most of the domains in the system. An available domain is not occupied by any thread. A domain becomes available when it performs a *return* operation.

Running Once it has been invoked, a domain moves from the available state to the *running* state. A running domain is busy servicing an invocation, and is occupied by a thread. If a start key is invoked while the domain is running, the invoker will block until the domain becomes *available*.

Waiting A domain that has invoked a key and is waiting for a response moves from the *running* state to the *waiting* state. It remains in this state until its current resume key is invoked. If a start key is invoked while the domain is waiting, the invoker will block until the domain becomes *available*.

3.1.3 Domain Keepers

As a domain interprets a program, it may take **execution faults** due to invalid or privileged instructions, inappropriate runtime data values, or execution of a trap instruction. Except in the case of the key invocation trap, the EROS kernel does not directly resolve such faults. Instead, the kernel encapsulates the fault into a message and delivers this message to the domain's keeper. The keeper is responsible for deciding what to do.[5]

Execution faults are distinguished from access faults. **Access faults** include references to invalid pages and accesses to pages mapped with insufficient access rights. If a *segment keeper* (see below) is defined, access faults will be delivered to the segment keeper in preference to the domain keeper. The segment keeper receives sufficient authority to pass the fault message back to the domain keeper for resolution.

Note that both access and execution faults are *endogenous* faults. Both can be traced directly to some action by the program. It is the responsibility of the keeper(s) to handle endogenous faults. Exogenous faults, such as memory errors, are the responsibility of the EROS kernel. A domain that takes an endogenous fault without a keeper defined becomes **broken**, and ceases to execute instructions.

The essential notion behind keepers is that whatever may prove to be wrong, the kernel lacks sufficient information to correct the problem. It is better to place fault handling in a user domain which might be able to do something constructive to address the problem.

3.2 Segments

A **segment** provides a mapping from offsets to pages (Figure 4). Segments serve as both the files and the address spaces of the EROS system. Segments are tree-structured, following the style of traditional tree-structured mapping tables. Each layer in the segment

[4] EROS keys will shortly be growing from 3 words to 4. The size of the number key payload will grow accordingly.

[5] A notable exception is software emulation of unimplemented instructions or unimplemented boundary conditions, which are performed transparently by the kernel in some implementations. Emulation could in fact be handled by the domain keeper, but this would place an unnecessary burden of implementation on all keepers and render them highly machine dependent. On balance, it appears that in-kernel instruction emulation is a better solution provided it is not excessive.

tree translates four bits of a segment offet.

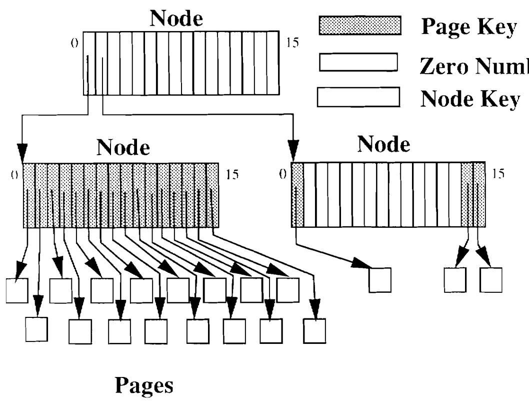

Figure 4: A 19 page segment

Every domain has an associated address space segment, which is referenced by load and store instructions. Programs access other segments by mapping portions of them into their address space segment.

3.2.1 Segment Keepers

Segment keepers subsume most of the memory policy management decisions of conventional kernels. Just as programs can execute invalid instructions, they can perform invalid references. A segment can optionally provide a *start key* to a **segment keeper**, a domain that should handle access faults encountered within that segment. This receives notification of both invalid accesses and protection violations.

A segment keeper can respond to a segment fault in one of two ways: it can modify the segment so as to allow the access to proceed, or it can decide that the reference is truly invalid and hand off the responsibility to the domain's domain keeper. Object cache misses are not reported to the segment keeper, but are transparently handled by the object cache.

3.3 Threads

A **thread** binds a domain to a particular machine. In this capacity, threads may be thought of as the roots of the state reachable from a given host. A running domain is occupied by a thread, and executes instructions on the host identified by the thread. A host implements a fixed supply of threads.

Threads are the unit of scheduling in the EROS abstract machine. When a domain calls another domain, the thread is said to *migrate* to the called domain.

3.4 Invocation and Messages

A program wishing to access an object must possess a key naming that object. Access is obtained by *invoking* the key with a message that is delivered to the object. An EROS message consists of an **order code**, exactly four keys, and a contiguous data string of up to 64 kilobytes.[6] Message responses contain a **return code**, four keys, and up to 64K of data. Key register zero of every domain always contains the null key. If sending or receiving four keys is not required for a given message, unneeded key arguments can be sent from or received to key register zero.

EROS supports three types of invocation: *call*, *fork*, and *return*. The **call** invocation passes a **resume key** in the last slot of the message (overwriting any key sent in that slot), and places the invoking domain in the *waiting* state until the generated resume key is invoked. The recipient object is placed in the *running* state as a consequence of the call. The **return** invocation is the inverse of the call operation. It places the invoker in the *available* state and the recipient in the *running* state. Resume keys are self-consuming: any invocation of a resume key causes all copies of that resume key to be efficiently invalidated. This ensures that every call receives at most one reply.

The **fork** invocation transfers the message to the recipient and leaves both sender and receiver in the *running* state. Unless the caller makes explicit provision for a reply, no response to a fork is possible.

Because domains are single-threaded, entry to a domain provides an implicit guarantee of mutual exclusion. This fact is exploited by a number of different EROS objects.

4 Realizing the Machine

The abstract machine must be mapped onto the underlying hardware. This is principally a matter of constructing the appropriate mapping table structures and finding a way to structure context information to facilitate efficient context switching. EROS views the mapping tables as a cache of the state embodied in the segment structures. Any domain state that is needed for efficient context switching is main-

[6]The current implementation restricts messages to be no larger than the architecture's page size. We plan to extend message data payloads to 64K shortly.

tained in the **context cache**. For both caches, dependency structures are maintained that allow the caches to be invalidated as objects are modified or removed from memory.

The design of the dependency mechanisms must meet three objectives:

1. New dependency structures must be constructable without long delay, if necessary by reclaiming other dependency-related data structures.

2. It must be possible to reclaim dependency structures incrementally.

3. The dependency structures must facilitate an efficient realization of ageing and checkpointing.

In this section we present the dependency tracking mechanisms that EROS uses to realize the abstract machine.

4.1 Key Slot Hazards

Because a key's value may be cached, care must be taken when reading or writing key slots to ensure that any dependent cache information is appropriately flushed and/or updated. Such a condition is known in EROS kernel terminology as a **hazard**. Hazards are divided into two flavors: read hazards and write hazards.

A **write hazard** indicates that the content of some cache depends on the current value of the key occupying a slot, and the cache must be flushed before a slot write can safely proceed. Write hazards arise in segment trees, where mapping entries may need to be invalidated, and in domains, where writing a slot may require updating the context cache structure associated with the domain.

A **read hazard** indicates that the current content of the slot is not up to date, and must be flushed back to the key from a cache. Read hazards arise in domains, where up-to-date register values may need to be flushed back to the domain from the context cache. All read hazards are also write hazards.

On multiprocessors, both read and write hazards may imply the need for interprocessor signalling to arrange for translation caches and/or register sets to be flushed.

4.2 The Context Cache

While number keys provide a space-efficient storage medium for register values, they are not an especially efficient format for loading and saving registers during a context switch. For this reason, the register values of a domain are loaded into a machine-specific **context** structure before the domain context is loaded onto the hardware. The domain layout is chosen for the convenience of the abstract machine. Because cross-domain message passing and context switching performance is critical, the context structure layout is chosen for efficient context switch. A number of implementation tricks are used to facilitate rapid context structure save and restore.

In conventional operating systems, processor state is saved to an interrupt stack, and later transferred to a per-process structure. In EROS, we contrive for the initial interrupt stack pointer to point to the top of the context structure for the active domain. We save the process state directly into the context structure and then switch interrupt stacks. As a result, the domain state is saved into the process structure without the need for a later copy. This technique is similar to techniques used in L3 [Liedtke93] and Mach 4.0 [Ford93].

Because EROS programs can hold node keys to the components of domains, however, we must be able to efficiently and selectively flush subsets of a context cache entry back to the constituent nodes of the domain on demand. In clearing hazards associated with key slots in the domain root, portions of the context cache entry may be unloaded. Before running a process or examining its context structure, the kernel must first verify that the domain is fully loaded into the context. For efficiency in context switching, a simple zero-test of the context save area field is sufficient to determine if the context is fully cached and runnable.

4.3 Mapping Table Management

When a translation fault occurs, the program's address space segment is traversed by the kernel to construct an appropriate mapping table entry. Invalid offsets and access rights violations are encapsulated by the kernel and reported to a user-level fault handler. The hardware mapping tables, in effect, are a lazily-generated projection of the access rights conveyed by the segment.

There are three circumstances under which a mapping table entry can cease to be valid:

1. The content of a page frame named by the mapping entry can be removed from memory.

2. A key slot traversed in the construction of a mapping entry can be overwritten with a different key.

3. A node containing a slot traversed in the construction of a mapping entry can be removed from memory.

Each of these cases must be addressed by supporting dependency management structures.

4.3.1 The Page Dependency Cache

Page reclamation is addressed by maintaining an inverted page table. Whenever a page table entry is created that points to a page in the object cache, a dependency entry is added to the *page dependency cache*. The page dependency cache is indexed by a hash on the page frame address. A three-sided data structure is constructed, as shown in Figure 5. When the page frame is reclaimed, all mapping entries naming that page frame are located via the page dependency cache and invalidated. Before any page dependency cache entry is flushed, the corresponding mapping entry is invalidated.

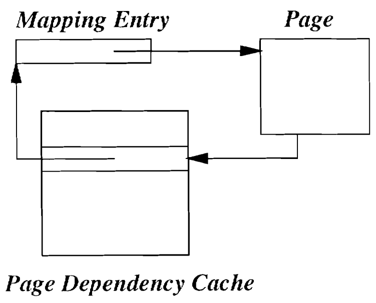

Figure 5: Page dependency structure

As with the object table, no attempt is made to update the page dependency cache when a mapping page is discarded by the ager. It is the responsibility of the mapping invalidation logic to verify that the mapping entry pointer points to a valid mapping entry that should be invalidated. On architectures using hash structured or software-managed mapping structures, this is not difficult; mapping structures on these machines are taken from a reserved pool of memory. On machines using page-sized mapping tables, mapping page frames are allocated out of the page frame cache. Before invalidating the mapping entry on such machines, the dependency cache must verify that the mapping page frame has not been reclaimed for other purposes.

Note that the hardware mapping table structures are a cache of the state captured by a segment tree. Mapping tables can always be discarded completely with no loss of information. If this is done, they will be rebuilt as necessary.

4.3.2 The Slot Dependency Cache

Node removal, surprisingly enough, is more straightforward. All of the segment-related dependencies in a node are covered by the dependencies on the node's slots. If all of the key slot dependencies for the slots in the node have been invalidated, the node can safely be removed. This reduces the problem to the tracking and management of key slot dependencies.

To understand how key slot dependencies are tracked, it is useful to see how they are constructed by the segment traversal algorithm. A simplified version of the traversal algorithm is shown in Figure 6. The crucial line is the call to `Depend::AddKey()`. This builds an inverse projection from the mapping table entries back to the key slots that generated them, allowing the dependency entries to be found and invalidated when the slot is overwritten or its containing node is removed.

As each slot is traversed on the path to constructing the mapping entry, the traversed slot is marked as write hazarded. To clear the write hazard, the associated slot dependency cache entries must be invalidated.

4.3.3 Merging Related Entries

A typical slot in a segment tree is involved in the generation of multiple mapping entries. On machines with tree-structured hardware mapping tables, these entries tend to be contiguous within a common mapping page. On such machines, the slot dependency cache will coalesce the shadowed mapping entries into a common cache entry describing a set of adjacent mapping entries.

This optimization proves to be an important source of space efficiency. An open issue in the current design is how to accomplish similar efficiencies on machines with hashed mapping tables or software-based miss handlers.

```
while (depth < MAX_DEPTH) {
  Word segBlss = pSegKey->blss;
  if (pSegKey->readOnly && writeAccess)
    SEGFAULT(FC_SegAccess);

  pSegKey->Prepare();
  Depend::AddKey(pSegKey, thePTE);

  // The last key might be a page key:
  if ( pSegKey->IsSegKeyType() )
    pSegKey->pObject->PrepAsSegment();

  if (segBlss <= target_blss)
    return pSegKey;

  // Traverse this node:
  Word shiftAmt = segBlss * 4;
  Word ndx = address >> shiftAmt;
  ndx &= 0xfu;

  pSegKey = &(*pSegKey->pObject)[ndx];

  depth++;
}
```

Figure 6: The traversal algorithm

5 The Checkpoint Mechanism

EROS implements persistence and exogenous fault recovery using the same mechanism: a recoverable checkpoint using a circular checkpoint log. The checkpoint mechanism is similar to that of KeyKOS [Landau92], but the use of a circular log makes it more adaptable to runtime load variations.

Before any object may be modified in memory, space is reserved for it in the log. When the dirty object is later written to the disk, it is appended to the log. Object cache misses are satisfied from the log if the object is found in the checkpoint log catalog, or from the object's home location if it is not (Figure 7).

Periodically, or when the available checkpoint log space reaches a low watermark, the kernel declares a checkpoint. When a checkpoint is declared, the system is frozen and all dirty objects are flushed to the log. The current log catalog is then flushed to the log along with the current thread list, and the checkpoint log header is revised to give the location of the most recently committed catalog. At this point the checkpoint has completed successfully, and a migrator is started to copy the objects back to their home locations. When the migrator has completed its migration, it updates the checkpoint log header

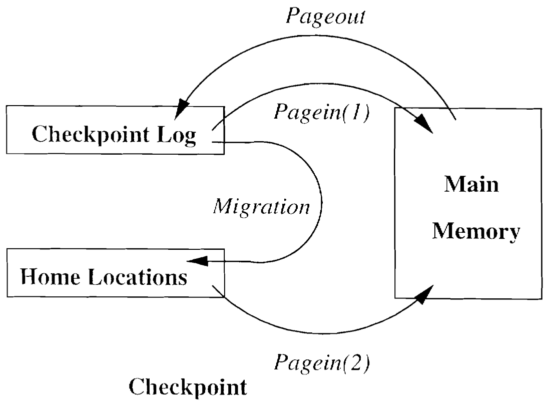

Figure 7: Flow of objects in the system

to indicate that space in the log previously occupied by the checkpointed state is now free. The net effect is that the system is always able to recover from the most recent successful checkpoint.

While conceptually accurate, the algorithm just given would be intolerably slow in practice. Flushing the dirty objects takes a significant amount of time, and delays as short as 100 ms are noticable to users in the form of mouse jitter and character echo stalls. To avoid long delays, checkpoint is performed in three phases.

5.1 Freezing the Image

The first phase of the checkpoint is to freeze all dirty objects in memory, ensuring that they will not be modified until the checkpoint has completed. Objects are frozen by marking them as write hazarded, and disabling the write permission bits on all page table entries. Finally, all user threads are deprepared, and a frozen copy is made of the in-core thread list. Execution is now permitted to resume. As each mutating reference is performed, the associated objects are duplicated using a copy-on-write mechanism.

As with key slots, hazards are set on an object frame when that frame is involved in I/O. During inbound I/O, there is a brief window of time during which the object frame exists but the inbound data transfer is incomplete. During this window, the object is read hazarded. Similarly, the object is write hazarded during on outbound I/O. Objects hazarded by I/O have a non-zero I/O count in their object header.

The 100 ms target allows us to freeze up to half a gigabyte of main memory with no noticeable delay, and perhaps 128 times that much by rearranging the hazard bits to favor cache locality. Ultimately, hierarchical or watermarking techniques can be used to scale this phase further.

5.2 Writing the Dirty Objects

Once the mark phase has completed, the object write phase can proceed at low priority. Objects marked for checkpoint are incrementally written to the checkpoint log, at a pace sufficient to complete the process well before the next checkpoint occurs. No special support is needed to cause this to happen. The cleaning daemon simply pages the objects out, decrementing the number of pending checkpoint I/O's as it goes along. When the count reaches zero, it has successfully written all of the checkpoint state.

When all of the dirty objects have been paged out, the thread list and the checkpoint catalog are written out to the checkpoint log. While space for the objects in the log is prereserved, the placement of any given object in the log is not known until the page daemon forces that object to the disk. Writing the catalog must therefore be delayed until all objects have been written.

Finally, the checkpoint log header is updated to reflect the location of the catalog associated with the most recently completed checkpoint. To guard against write failure, the checkpoint log header is written sequentially to two different sectors. This ensures that one copy of the log header always has valid data.

5.3 Migration

Once a checkpoint image has been completely written to the checkpoint log, a migration phase must be started to free space in the log for the next checkpoint. The migrator examines the checkpoint catalog (which remains in memory) to determine an efficient order of operation, and copies the objects from the log to their home location. In practice, many of the objects that need to go back to home locations are still in memory, and these objects do not need to be read back in from the log.

One advantage to the circular log structure is that the migrator can proceed incrementally [Gray93]. If there is significant pressure on the log, the migrator can move a small number of objects, write a new checkpoint catalog, and update the checkpoint log header to reflect the new "most recent checkpoint" catalog.

6 Conclusions

The EROS system is currently running on i486 and Pentium class hardware. The implementation requires less than 64 Kilobytes of supervisor code, made up of approximately 18,000 lines of C++ source and 1400 lines of assembly code. The C++ code is evenly split between machine dependent and machine independent code. Our intention is to make the implementation widely available when completed. To be placed on the announcement list, please visit the EROS home page at `http://www.cis.upenn.edu/~eros`

We will shortly be turning our implementation efforts to the distributed version of the system.

6.1 Lessons Learned

The design described here reflects knowledge gained from several false starts, and several departures from earlier work.

Threads

The KeyKOS system [Hardy85] had no first-class notion of a thread, and nothing described in this paper motivates their inclusion in the abstract machine. Threads were originally introduced in the EROS kernel to support prioritized 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.

Once threads were included in the architecture, however, they provided a natural way of specifying what host a domain should compute on in a cluster. The rule is that objects migrate to the requesting thread. For this reason, they have been retained.

Capability Encryption

A number of prior capability systems have adopted encryption as the mechanism for capability protection. Encrypted capabilities allow the capability tokens to reside within the user address space, eliminating the need for direct kernel management. For several reasons, we have chosen not to use this mechanism.

Encrypted capabilities rely in part on the ephemeral nature of the holder to defeat forgery attacks. In most encrypted capability systems this issue is ignored. In a few, time stamps are used to ensure that capabilities are ephemeral. The EROS system is persistent, which violates the ephemerality assumption. Perhaps more important, encrypted capabilities would not lend themselves well to the construction of segments and domains out of fixed-size fundamental objects, nor to the maintainance of cache consistency via hazard tracking.

Node Size

A crucial part of EROS's simplicity and small size is that the kernel synthesizes all of the abstract machine's objects out of two fundamental object types. This eliminates the need for the kernel to do dynamic storage management, which improves both its performance and its reliability. Objects on the disk are of only two types, which substantially simplifies the implementation of the single level store.

At the suggestion of Bryan Ford, we briefly considered following the path of L3 [Liedtke93], which captures all system state in objects of the same size. This would increase the node size to 256 slots. Our initial reaction was that this would be unduly wasteful, but in thinking this we may have been mistaken. The page-sized node is not significantly larger than the current domain of three nodes, and might well prove better suited to the construction of small segments, many of which are over 16 pages and therefore occupy three or four of the current nodes in any case. With the benefit of hindsight, the slight loss in space efficiency might be more than recouped in simplifications of the dependency management structures and the checkpoint mechanism. It would also allow the kernel to respond gracefully to dynamic variations in page pressure versus node pressure.

6.2 Acknowledgements

In pursuing this work, we are deeply indebted to the assistance of far more people than can reasonably be acknowledged. 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 KeyKOS group.

Early readers caught a number of errors – some substantial – in this paper. Thanks to Mitchell P. Marcus, William Frantz, and Colin McLean for their efforts in reviewing under severe time pressure.

References

[Bomberger92] Allen C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, Jonathan S. Shapiro. "The KeyKOS NanoKernel Architecture," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association. April 1992. pp 95-112.

[Ford93] Bryan Ford and Jay Lepreau. "Evolving Mach 3.0 to a Migrating Threads Model." *Proceedings of the Winter USENIX Conference*, January 1994.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing, Concepts and Technology.* Morgan Kaufmann. 1993.

[Hardy85] Norman Hardy. "The KeyKOS Architecture" *Operating Systems Review*. Oct. 1985. pp 8-25.

[Landau92] Charles R. Landau. "The Checkpoint Mechanism in KeyKOS," *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* IEEE. September 1992. pp 86-91.

[Liedtke93] Jochen Liedtke. "Improving IPC by Kernel Design," *Proceedings of the 14th Symposium on Operating System Principles.* Asheville, N.C., IEEE, 1993.

[Shap96a] Jonathan S. Shapiro. *A Programmer's Introduction to EROS.* Available via the EROS home page at `http://www.cis.upenn.edu/~eros`

[Shap96b] Jonathan S. Shapiro. *The EROS Object Reference Manual.* In progress. Draft available via the EROS home page at `http://www.cis.upenn.edu/~eros`