An Introduction to Mach 3.0's XMM Subsystem

Bill Bryant (billb@osf.org)
Steve Sears (sjs@osf.org)
David Black (dlb@osf.org)
Alan Langerman (alanl@osf.org)
OSF Research Institute
Cambridge, MA
June 14, 1993

Abstract

The XMM subsystem is an extension to the Mach 3.0 kernel that enables the kernel to provide coherent distributed shared memory in multicomputer environments. Currently, XMM implements a strict multiple readers/single writer sharing policy. After presenting a brief review of Mach virtual memory management, Mach IPC and MIG, and the External Memory Manager Interface, this paper presents a conceptual description of XMM's object-oriented framework.

1 Introduction

The Mach 3.0 microkernel provides a flexible execution environment for both system and user applications. It provides a set of low-level operating system primitives that can be used to implement different operating system personalities.

The XMM subsystem is an extension to the Mach 3.0 kernel that enables it to provide coherent distributed shared memory in multicomputer environments. (XMM is an acronym for *eXtended Memory Management*.) XMM also provides copy-on-reference and copy-on-write mechanisms that allow tasks on separate nodes to share copies of memory efficiently.

A multicomputer is a computer that can contain hundreds, even thousands of nodes. Each node includes one or more processors and the nodes communicate with one another via an interconnecting network. XMM allows programs executing on separate nodes to share read/write access to memory.

Multicomputers are categorized by the degree in which the nodes are able to access one another's resident memory. Machines whose nodes are not capable of accessing the memory of other nodes are referred to as a NORMA (NO Remote Memory Access) computers. In contrast, the nodes on NUMA (Non Uniform Memory) computers are able to access one another's memory, but the speed at which one node can access another node's memory depends on the distance that separates the nodes. XMM is designed primarily for NORMA computers.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

NORMA architectures range from super computer class multiprocessor machines in which thousands of nodes are connected to one another with a very high speed interconnect, to clustered configurations in which multiple desktop workstations are connected to one another via a standard Ethernet. XMM provides coherent shared memory for all types of NORMA architectures.

XMM implements a strict multiple readers/single writer sharing policy; multiple tasks may read the data at the same time, but only one task may write the data at any one time. If other tasks are reading the data when a task attempts to write the data, XMM first revokes the readers' access to the data, then allows the writing task to proceed. This is the only sharing policy that XMM currently supports.

2 A Quick Review of Mach and Mach Virtual Memory Management

The Mach microkernel implements a small set of primitive operating system objects, including tasks, threads, and memory objects. These objects can be used to support different operating system personalities. For example, the OSF/1 operating system uses Mach objects to support a UNIX environment.

A traditional UNIX process is a single entity that encapsulates a set of system resources (memory resources, open files, and so on), and a single thread of control that executes in the context of those resources. In Mach, the process abstraction is split into two separate abstractions: the task and the thread.

A task is a set of system resources that includes a protected virtual address space. A task is not in itself an executable entity; it is an environment that supports the execution of one or more threads. However, it is conventional usage to refer to tasks as entities that "do" things. This document employs that convention.

2.1 Task Address Spaces

A task's address space contains *regions* of allocated and unallocated memory. Generally, an allocated region of memory contains a specific type of data. For example, a task that constitutes a UNIX process contains four regions of allocated memory: a *text* region, an *initialized* data region, an *uninitialized data* region, and a *user stack* region.

The Mach kernel implements operations on a task's address space by maintaining an address map for each task. A task's address map associates each allocated region of memory with a *memory object.* A memory object represents the source of the region's data. For example, a region that contains program text is mapped to a memory object that represents the executable file in secondary storage.

Like UNIX processes, Mach tasks inherit their virtual address spaces from their parent tasks. However. under UNIX, a child process always inherits an entire copy of its parent's address space; the parent cannot selectively prevent its children from inheriting portions of its address space. Mach allows tasks to control the inheritance of address space. A child task may inherit all, some, or none of its parent's address space.

The inheritance mechanism allows a child task to receive a copy of the parent's region, or to share the region with the parent. If a child task inherits a copy of a region, the parent task will not see any of the child's modifications to the region. If the region is shared between the parent and child, each task will see modifications made by the other.

2.2 Memory Objects and Memory Managers

A memory object is an entity that represents a contiguous set of virtual pages. A task's address space is implemented by an address map that maps ranges of address space to specific memory objects. For example, a task that constitutes a UNIX process maps three memory objects into its address space: one that represents the pages containing the text and initialized data from the executable file, one that represents the zero-fill pages containing uninitialized data and heap, and one that represents the zero-fill pages containing the user stack.

Until recently, UNIX-based systems have not allowed processes to map arbitrary objects into their address spaces, and those UNIX systems that have allowed such mapping operations restrict them to kernel-defined objects such as files. The memory object abstraction allows developers to implement paging managers that manage application-specific memory objects. These objects may represent files, shared libraries, and databases.

Memory objects are managed by separate tasks referred to as memory managers or paging managers. The memory managers are responsible for performing pagein and pageout operations on the memory objects they manage. For example, when a task generates a page fault while attempting to reference a page that is not resident, the kernel issues a request to the appropriate memory manager to furnish the page. If, at a later point, that page becomes modified and the kernel's page replacement algorithm requires that the physical page be reallocated, the kernel issues a request to the memory manager to write the page to secondary storage.

2.3 Mach IPC

In Mach, tasks interact with their environments by sending messages to and receiving messages from other tasks. The facility that implements message passing is referred to as Mach IPC.

Messages are passed between tasks via communication channels called ports. Ports are unidirectional. In order for two tasks to pass messages back and forth, there needs to be two ports: one for each task to receive messages on.

Ports are protected by the kernel and so provide a secure mechanism for naming kernel-implemented and user-implemented objects. For example, when the kernel creates a new task, it allocates a port to represent the task. If another task has a *send right* to that port, it may invoke operations on the task object by sending messages to the port. Multiple tasks may have send rights to a port, but only one task may hold the *receive right* to the port.

The IPC facility is often used to implement client-server applications. On behalf of its clients, a server creates and manages objects. These objects are represented by ports. Typically, a client invokes operations on an object by sending a message to the server on the port that represents the object. These invocations are referred to as *remote procedure calls* (RPCs). In Mach, RPCs are implemented as follows:

- The client composes a message and sends it to the server on the object's port.
- The server receives the message and performs the operation on the object.
- The server composes a message indicating the success or failure of the operation and sends it to the client on a port the client has receive rights to.

Memory objects are represented by ports, and the interactions between a kernel and a memory manager are conducted using RPCs. For example, when a task generates a non-resident page fault in a particular memory object, the page fault handler requests the page by issuing an RPC to the memory manager via the memory object's port. The memory manager responds to the page fault request by sending the page to a port that the kernel holds receive rights to.

2.4 The Mach Interface Generator (MIG)

A client-server application requires three sets of routines. On the client side, the application must provide the RPCs that allow clients to request the creation and manipulation of server objects. On the server side, the application must provide the other two routine sets: the routines that create and manipulate the objects, and a set of stub routines that provide the interface between the server's internal routines and the client RPCs.

The implementation of client RPCs and server stub routines requires large amounts of code that performs message assembling and message unpacking. The Mach Interface Generator can be used to create this code automatically. For example, kernels and memory managers interact with one another via the External Memory Manager Interface, a set of routines that are generated by MIG.

2.5 The External Memory Manager Interface

Mach defines an External Memory Management Interface (EMMI) that the kernel and memory managers use to interact with one another.The EMMI is actually two sets of routines: those implemented by memory managers to be used by the kernel and those implemented by the kernel to be used by the memory managers.

The *memory object management interface* is the set of routines the memory manager provides so that the kernel can request operations on the manager's memory objects. This interface includes the following calls:

- **memory_object_init( )** - Requests the memory manager to initialize the memory object for mapping into a task's address space. This routine establishes communication between the kernel and the memory manager; the kernel allocates a control port for the memory object that the memory manager can use to send cache management requests to the kernel.

- **memory_object_terminate( )** - The kernel calls this routine when the memory object is no longer mapped by any of the tasks currently supported by that kernel and the kernel wishes to evict the object.

- **memory_object_data_request( )** - Reads the contents of a memory object. The kernel calls this routine to resolve page faults on the memory object.

- **memory_object_data_return( )** - Writes modified pages back to the memory object. The kernel calls this routine to page out the memory object's contents.

- **memory_object_data_unlock( )** - Requests that the memory manager give the kernel permission to change the protection of the memory object's contents. For example, the kernel would call this routine if a task wanted to write data that was currently read-only.

The *cache management interface* is the set of routines the kernel implements to allow memory managers to maintain the contents of memory objects that are cached in resident memory. These routines include the following:

- **memory_object_ready( )** - Reply to **memory_object_init( )**; the memory manager uses this call to tell the kernel it is ready to perform operations on the specified memory object.

- **memory_object_data_supply( )** - Supplies the requested page of data. This routine complements the **memory_object_data_request( )** routine; the memory manager uses this routine to provide data in response to a page fault.

- **memory_object_data_error( )** - Indicates to the kernel that the kernel's request for access to data cannot be granted. The data may not exist, or the kernel may be attempting to access it in a way that violates its current level of protection.

- **memory_object_lock_request( )** - Requests that the kernel perform one of the following actions on pages cached in resident memory: flush the pages, return the pages to the memory manager, or change the protection value of the pages.A memory manager can use this routine to maintain consistency of data on multiple hosts.

2.6 An Example Shared Memory Manager

This section illustrates how a memory manager and two kernels would use the EMMI to implement network shared memory. The memory manager allows tasks on separate machines to share read/ write access to its memory objects, serializing write access to the data by allowing only one task at a time to write the data, and allowing multiple tasks to read the data when a write operation is not underway.

In Figure 1, two tasks running on separate machines have mapped the same memory object into their address spaces. When the clients are sharing read access to the memory object, the data is protected read-only on both machines. The figure shows how the memory manager interacts with the machines' kernels when client task 1 attempts to write the data:

1. Client Task 1 generates a protection page fault attempting to write a page that is protected read-only.

2. Kernel 1 sends a message to the shared memory server to request that Client Task 1 be granted write access to the page by calling **memory_object_data_unlock( )**.

3. The memory manager sends a message to Kernel 2 to invalidate the page cached on that kernel by calling **memory_object_lock_request( ).**

4. When Kernel 2 flushes the page, it sends a message to the memory manager by calling **memory_object_lock_completed( )**.

5. The memory manager sends a message to Kernel 1 indicating that the protection of the page can now be changed by calling **memory_object_lock_request( )**. Client Task 1 can now write the page.

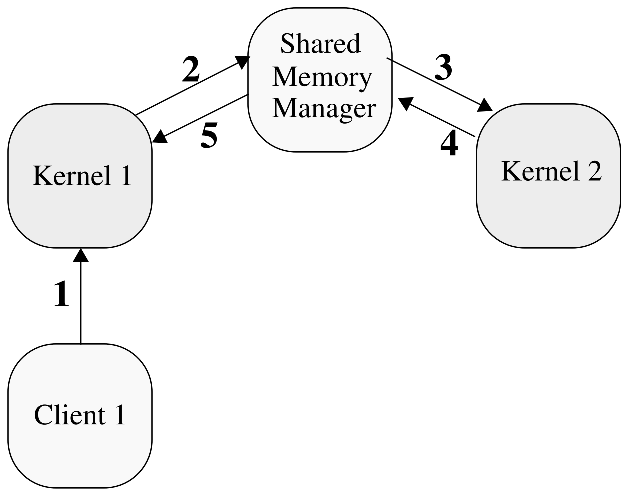

Figure 1. Client 1 Writes the Data

Suppose that Client Task 2 (running in Kernel 2) subsequently attempts to read the page that is being written. As shown in Figure 2, Client Task 2 initiates the following operations:

1. Client Task 2 generates a page fault because the page was flushed when Client Task 1 attempted to write it.

2. Kernel 2 sends a message to the memory manager requesting a pagein operation by calling **memory_object_data_request( )**.

3. The memory manager, via a call to **memory_object_lock_request( )**, sends a message to Kernel 1 indicating that the kernel should pageout the page.

4. Kernel 1 returns the page by calling **memory_object_data_return( )**, then calls **memory_object_lock_completed( )** to indicate that it has responded to the memory manager's lock request.

5. The memory manager now has a valid version of the page; the server provides the page to Kernel 2 by calling **memory_object_data_supply( )**.

6. The kernel allows the client to resume execution.

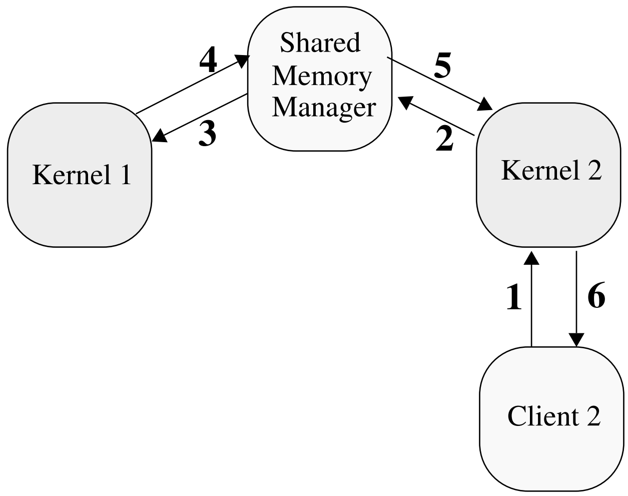

Figure 2. Client 2 Reads the Data

3 Extensions to Mach for Multicomputer Environments

Two major kernel subsystems--XMM and NORMA IPC--have been added to Mach 3.0 to enhance its suitability for multicomputer environments. XMM, in addition to providing distributed shared memory, also includes support for copy on reference between nodes for lazy evaluation of memory inherited between tasks on different nodes.

NORMA IPC provides location-transparent communication that takes advantage of the high bandwidths and low hardware latencies and software latencies inherent in multicomputer interconnects. In earlier Mach systems, the IPC's location-transparent mechanisms were extended across local area networks via user-level communication servers such as the *Net Message Server*. NORMA IPC provides a super set of the fuctionality provided by the Net Message Server. It allows kernels on separate nodes to communicate directly with one another. XMM uses NORMA IPC to communicate with instances of itself on separate nodes.

4 XMM as a First-Level Pager

In the example described in section 2.6, the memory manager treats each kernel as a separate cache of its pages. To maintain a memory object's global state, the memory manager must keep track of the disposition of all of the pages cached on the different kernels.

There are some disadvantages in such an arrangement. For example, managing the contents of a memory object that has pages cached on different kernels introduces a substantial level of complexity into the memory manager. There are also performance considerations. Each operation on a memory object's pages requires the attention of the memory manager, which executes in user space.

Under XMM, a kernel does not have to go to the memory manager to get a page if that page has been cached by a kernel on another node. Instead, XMM can fetch the page directly from the other node. In this respect, XMM acts as a first level pager. The memory manager becomes involved in the page fault path only if the page isn't currently cached on some node.

XMM is part of every Mach kernel on a multicomputer. With XMM in place, memory managers can interact with the multicomputer as a single cache. This removes the complexity of distributed shared memory functionality from memory managers.

5 XMM Framework

The XMM subsystem sits between the multicomputer's kernels and its memory managers. As in other Mach systems, kernel and memory manager still invoke operations on a memory object using the External Memory Manager Interface; however, in a multicomputer environment, XMM transparently intercepts all memory manager and kernel calls to the EMMI.

In addition to interacting with the kernels and memory managers, XMM must interact with itself. For example, when a given memory object is mapped on separate kernels, the instances of XMM running on those kernels must be able to communicate with one another. XMM uses the NORMA IPC subsystem to perform this communication.

Figure 3 illustrates the points of interaction that XMM must handle when a memory object is mapped on two separate kernels. In the figure, kernel A mapped the memory object first, and kernel B mapped the object later. As shown in the figure, the memory manager interacts with the instance of XMM on kernel A, the kernel that initially mapped the memory object, and the two instances of XMM interact with one another via NORMA IPC. In addition to maintaining the coherency of the memory object, XMM must also manage the interactions between itself, the kernels and the memory manager.

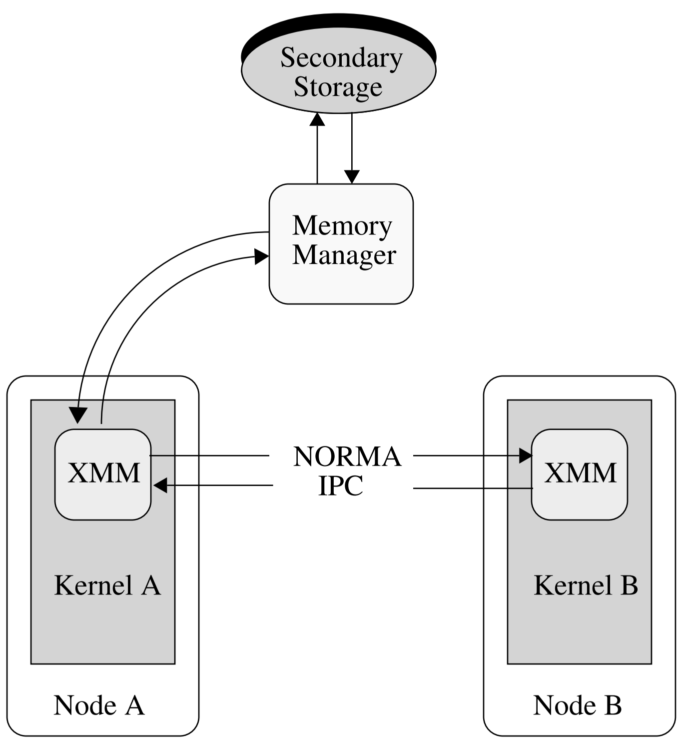

Figure 3. XMM Interactions

5.1 Object Stacks

XMM uses an object-oriented framework that divides the subsystem's interactions and shared memory operations into discrete *objects*. XMM maintains an *object stack* for each mapped memory object and uses this object stack to manage the various interactions and operations that may occur on the memory object.

An object stack is initialized when a previously unmapped memory object becomes mapped. Upon initialization, an object stack contains the following objects:

- *user object* - Handles interactions between the memory manager and XMM.
- *split object* - Manages multi-page requests on the memory object. See Section 7 for further discussion.
- *Shared Virtual Memory (SVM) object* - Implements coherent shared memory for the memory object.
- *server object* - Handles interactions between XMM and the kernel.

Figure 4 shows the object stack associated with a memory object that is mapped by a single kernel (kernel A). At this point, the entire object stack resides on the node that first mapped the memory object. As will be seen shortly, the object stack grows and becomes distributed as other kernels map the memory object. However, the initial stack objects--the user, split, and SVM object--reside permanently on the node that first mapped the memory object.

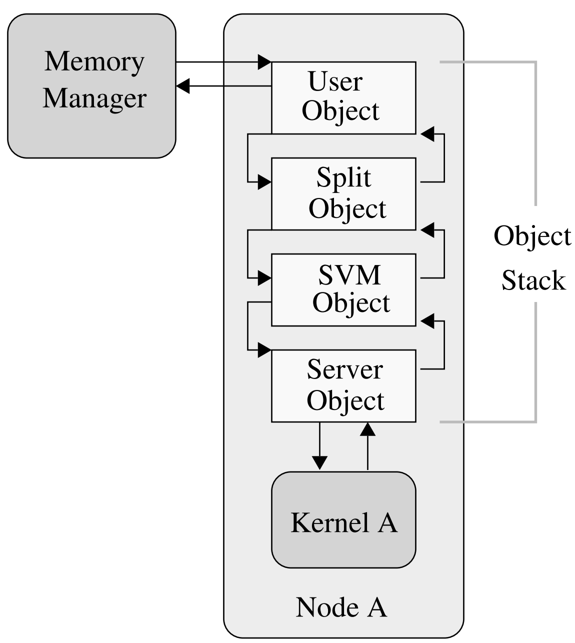

Figure 4. Initialized Object Stack

Figure 5 shows the object stack after kernel B maps the memory object. There is now a portion of the object stack that resides on kernel B's node. This portion of the stack contains a server object to manage XMM's interactions with kernel B. The stack also contains an *import object*, which matches the *export object* that has been added to the object stack on kernel A. These two objects manage the interaction between kernel A's and kernel B's instances of XMM.

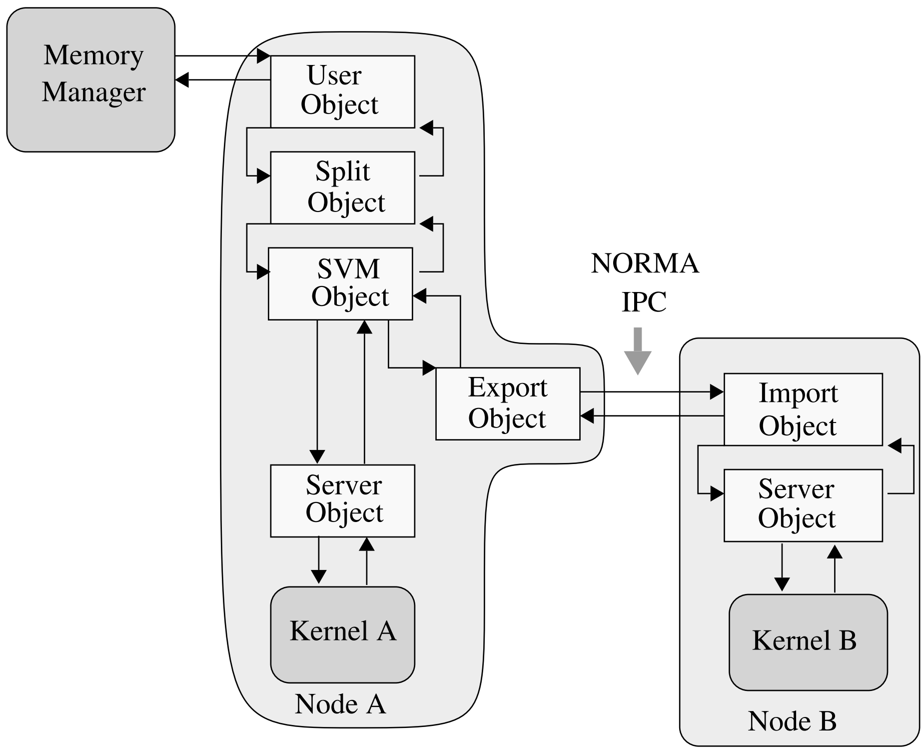

Figure 5. Object Stack on two Kernels

Figure 6 shows the object stack after kernel C maps the memory object. As with kernel B, the portion of the stack residing on kernel C's node contains a server object and an import object. Note as well that the object stack on kernel A has a new export object that compliments kernel C's import object. Each time a new kernel maps the memory object, XMM creates a new import/export object pair to handle interactions between the remote and local instances of XMM.

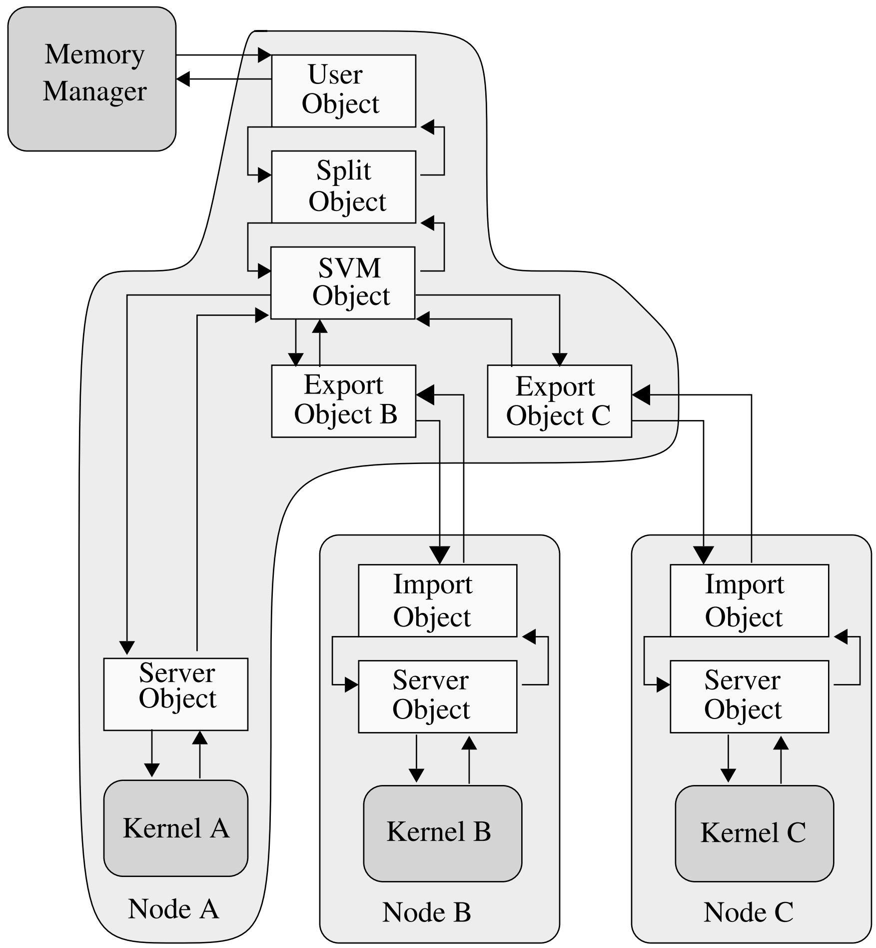

Figure 6. Object Stack on Three Kernels

6 The SVM Object

A memory object's SVM object maintains the state required to keep the memory object's contents coherent when it is mapped on multiple kernels. The SVM object is itself composed of multiple objects: one *Manager SVM object*, and one or more *Kernel SVM objects*.

The Manager SVM object maintains global state with respect to the memory object's pages by maintaining an array that identifies those pages that have been cached. When a kernel generates a cache miss fault, the Manager SVM object uses this array to determine where to get the page from.

A kernel SVM object maintains a memory object's state with respect to a specific kernel. This object contains a dynamically allocated array that specifies state for each of the memory object's pages that are currently cached by that kernel. The SVM object maintains a kernel SVM object for each kernel that has mapped the memory object. When the Manager SVM object determines that a particular page has been cached on some kernel, it iterates through the kernel SVM objects until it finds a kernel that has cached the page.

For example, Figure 7 shows the SVM object that manages a memory object which has been mapped by three separate kernels. Suppose the first four of the memory object's pages are cached on Kernel A, the first two are also cached on kernel B, and the first three are also cached on kernel C. Suppose that kernel C wants to cache the memory object's fourth page. When Node C's kernel generates a page fault and issues a call to **memory_object_data_request( )**, XMM determines via the manager SVM object that the page has been cached. XMM then checks each kernel SVM object to find the page. As soon as it has located a copy of the page, XMM initiates the appropriate set of operations to bring that page (or a copy of the page if the requested access is read-only) to the Node C kernel.

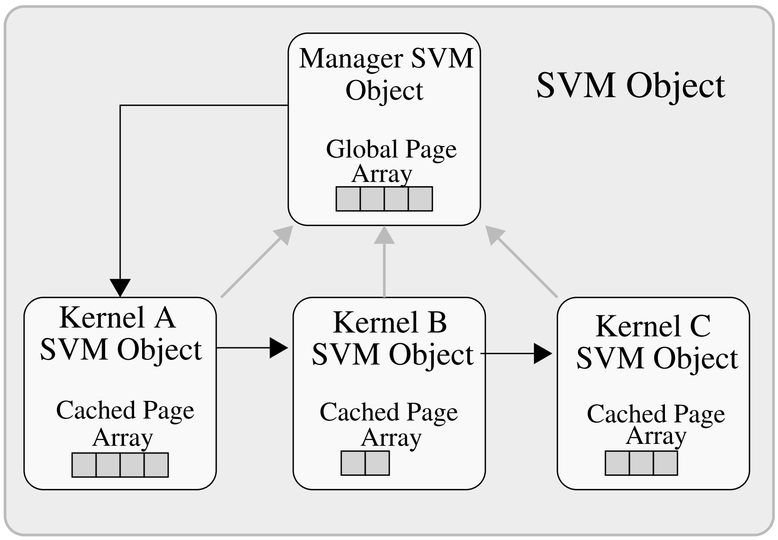

Figure 7. Manager and Kernel SVM Objects

As shown in Figure 7, the Manager SVM Object accesses the kernel SVM objects via a linked list, and each kernel SVM object includes a pointer to the Manager SVM Object.

7 The Server Object

The server object manages the interactions between the kernel and XMM (see Figures 4 - 6). It communicates with the kernel via the External Memory Management Interface. For example, when a kernel calls **memory_object_data_request( )** to resolve a not-resident page fault, the server object converts the call into an XMM request and invokes the request on the next object in the XMM object stack. When the requested page is found and makes its way back down the XMM object stack, the server object provides the page to the kernel by invoking **memory_object_data_supply( )**.

The server object also contains the routines that are used to initialize the object stack when the kernel maps the memory object. If the memory object being mapped is not mapped by any other kernel, the server object initializes a new object stack, which will reside locally. If the kernel is mapping a memory object that is mapped by another kernel, an object stack already exists on a remote node;

the kernel's server object adds itself to the object stack by initializing a local import object and an export object on the remote node.

8 The Export and Import Objects

The export and import objects are responsible for handling the XMM interactions that take place across node boundaries. As these interactions are conducted via NORMA IPC, import and export objects do nothing more that compose, send, and receive MIG messages, which are then sent and received via NORMA IPC.

Figure 8 shows the relation of export and import objects to the rest of the object stack. The figure shows a portion of the object stack for a memory object hat has been mapped on Kernels A, B, and C. The portion of the object stack that is specific to Kernel A does not have export and import objects because Kernel A is the local kernel. Kernels B and C are remote kernels; they mapped the memory object after Kernel A performed the initial mapping. As shown in the figure, each export object is positioned beneath the respective kernel's kernel SVM object. An object stack will include import and export objects only if the memory object is mapped on one or more remote kernels.

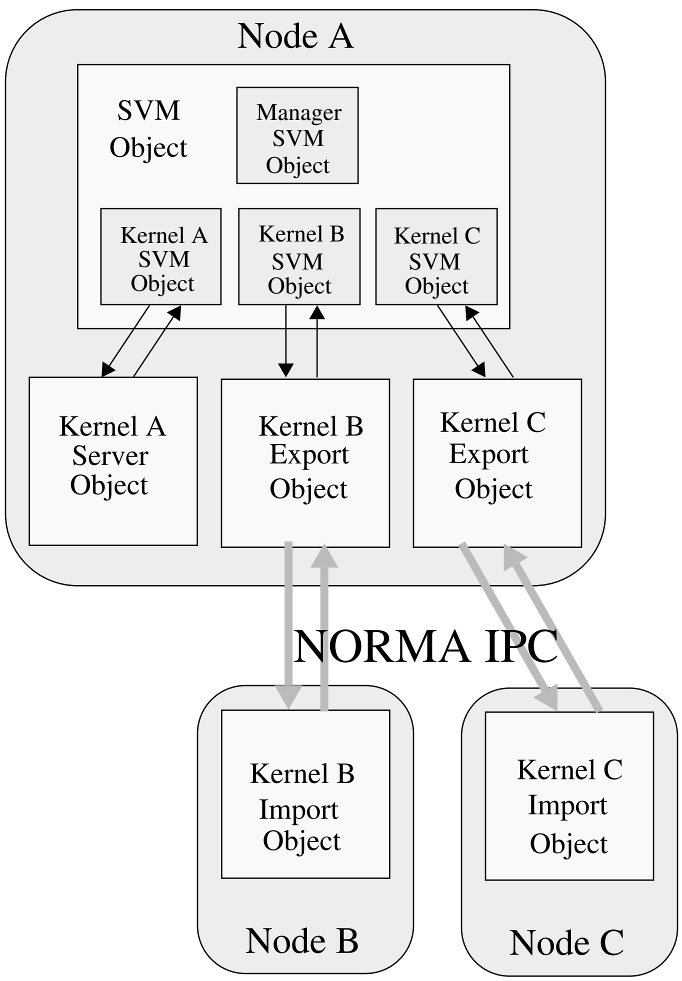

Figure 8. Export/Import Object Relationships

9 The User Object

The user object provides the transparent interface between the Memory Manager and XMM. When the memory manager uses the External Memory Manager Interface to request an operation on a memory object, the user object validates the request, converts it into an XMM method, and invokes that method on the split object, the next object down the object stack. When XMM replies to a memory manager's request, the user object converts the reply into a call to the External Memory Manager Interface.

10 The Split Object

The cache management portion of the External Memory Manager interface includes the **memory_-object_lock_request( )** call, which a memory manager may use to request that a kernel change the protection value of pages that kernel has cached. This call allows the memory manager to change the protection values on a contiguous set of virtual pages.

Under XMM, a memory object's pages may be spread across multiple nodes, and it may be that a given set of contiguous virtual pages are not cached together on any of the nodes. For example, suppose that a particular memory object is mapped on three nodes, and that the object's first page is cached read/write on node A, its second page is cached read/write on node B, and its third page is cached read/write on node C. Suppose that the memory manager issues a call to **memory_object_-lock_request( )** to change the protection of these pages to read-only.

The split layer intercepts this call and divides it into three separate requests. When each kernel has completed the lock request, it informs the memory manager by issuing a call to the **memory_object_lock_completed( )** interface. The split layer processes the individual lock_completed messages and coalesces them into a single message which it then forwards to the memory manager.