A Flexible External Paging Interface

Yousef A. Khalidi
Michael N. Nelson

SMLI TR-93-20 October 1993

Abstract:

In this paper we describe an aspect of the Spring virtual memory system that was influenced by the distributed object-oriented architecture of Spring. The virtual memory system supports external pagers like those provided in the MACH® operating system, yet the architecture is more flexible and provides better caching opportunities than is possible in other systems. A novel aspect of the architecture is the separation of the memory abstraction from the interface that provides the paging operations. This separation provides considerable caching opportunities in our file system, and it facilitates our extensible stackable file system architecture. The virtual memory architecture described in this paper is implemented and has been in use for over three years as part of the experimental Spring operating system.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
yousef.khalidi@eng.sun.com
michael.nelson@eng.sun.com

A Flexible External Paging Interface

Yousef A. Khalidi *Michael N. Nelson*

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

The Spring operating system is an experimental object-oriented operating system developed by our research group at Sun Microsystems Laboratories, Inc. In the Spring operating system, the object paradigm pervades and unifies the system. The system is structured around a small nucleus that provides the basic mechanisms for object invocation and thread control. Traditional operating system functionality (such as file system services) is built on top of the substrate provided by the nucleus as user-level applications. The Spring operating system is a distributed multi-threaded system that is constructed to exploit a range of systems from tightly-coupled multiprocessors to more loosely-coupled networks. The Spring operating system supports traditional UNIX® programs through compatibility mechanisms [1], but it is aimed at new computing requirements, such as transparent distribution, high reliability, and stronger security.

We designed and implemented a Virtual Memory System for the Spring operating system. The VM system follows the Spring object model and strives to meet the diversity of applications intended for the Spring operating system. The Spring VM system provides:

- Flexible, distributed, and secure memory mapping and sharing.
- Well-defined object-oriented interfaces for external (user-level) pagers.
- Support for distributed shared memory.
- Support for stackable file systems.
- Support for efficient bulk-data transfer mechanisms.

Spring is an internal code name only.

The design and implementation of the Spring VM system are described in [2]. In this paper we concentrate on one aspect of the VM system, namely the separation of the memory abstraction from the interface that provides the paging operations. This separation provides considerable caching opportunities in our file system and it facilitates our extensible stackable file system architecture.

This paper is organized as follows. The rest of this section provides a quick overview of the Spring system. Section 2 describes aspects of the VM system that are relevant to this paper, while Section 3 introduces the notion of separating the memory abstraction from the paging interface. Section 4 describes a protocol used between the VM system and external pagers. Examples of how we utilize the separation of memory from paging are described in Section 5. Section 6 compares our work to other external pager-based systems. Concluding remarks are offered in Section 7.

1.1 The Spring Operating System

The Spring operating system is a distributed, multi-threaded operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of the object and its operations is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) and the *client* of the object.

The Spring operating system strives to keep a clear separation between *interfaces* and *implementations*, and in

general, there is no special status for interfaces that are provided as part of the base system. The Spring system is perceived as a set of interfaces rather than a set of implementations.

A Spring domain is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the clients of other objects. The implementor and the client can be in the same domain or in different domains. In the latter case, the representation of the object includes an unforgeable nucleus *handle* that identifies the server domain.

Since the Spring operating system is object-oriented, it supports the notion of *interface* inheritance. An interface that accepts an object of type *foo* will also accept a subclass of foo. For example, the *address space* object has an operation that takes a *memory* object and maps it in the address space. The same operation will also accept *file* and *frame_buffer* objects as long as they inherit from the memory object interface.

The Spring kernel supports basic cross domain invocations, threads, and low-level machine-dependent interrupt handling, and provides basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a network proxy server. In addition, the VM system depends on external pagers to handle storage and network coherency.

A typical Spring node runs several servers in addition to the kernel (Figure 1). These include the domain and VM managers; a name server; a file server; a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a tty server that provides basic terminal handling as well as frame-buffer and mouse support. Support for running UNIX binaries is also provided [1] .

The Spring file system supports cache coherent files [3]. The file object interface inherits from the memory object interface and therefore can be memory mapped. The file system uses the VM system to provide data caching and uses the operations provided by virtual memory caches to keep the data coherent. The file system also acts as a system pager.

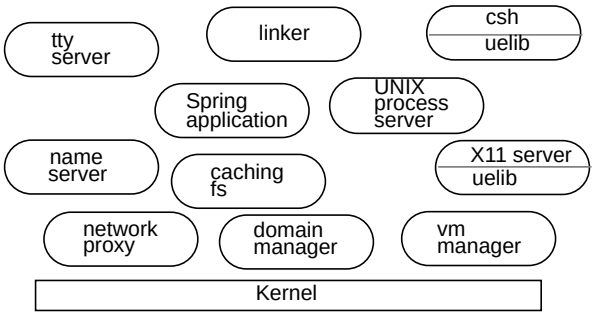

Figure 1. Major system components of a Spring node

2 Spring VM System

2.1 Overview

There are two sets of agents that cooperate to provide virtual memory in the Spring operating system. A per-node Virtual Memory Manager (VMM) is responsible for handling mapping, sharing, and caching of local memory. The VMM depends on external pagers for accessing backing storage and maintaining inter-machine coherency.

Most clients of the VM system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain, while a memory object is an abstraction of store (memory) that can be mapped into address spaces.

The main operations on address space objects are to map and unmap (part of) memory objects into selected address ranges of the address space. Since a memory object encapsulates a maximum access mode, a client may map a memory object as long as the requested access mode does not exceed the maximum access mode of the object.

A memory object has operations to set and query the length, and operations to *bind* to the object (see below). There are no page-in/out or read/write operations on memory objects (which is in contrast to systems such as Mach [4]). A holder of a memory object can either map it into an address space or pass it to another client. The significance of not providing paging operations on the memory object is explained in Section 3.

2.2 Cache and Pager Objects

The VM architecture defines two other types of objects: the *pager* object and the *cache* object. The pager object is implemented by external pagers. It provides operations to page in and out memory blocks and is used by the VMM to populate a local memory cache. The cache object is implemented by the VMM and is used by the external pager to affect the state of the cache. A given pager object–cache object pair constitutes a two-way communication channel between an external pager and a virtual memory manager. Typically, there are many such channels between a given external pager and a VMM. Tables 1 and 2 list the operations of the pager and cache objects, respectively.

Operation	Description
page_in	Request data be brought into the cache
page_out	Write data to pager and remove data from cache.
write_out	Write data to pager and retain data in read-only mode.
sync	Write data to pager and retain data in same mode.

TABLE 1. Pager object operations

The architecture defines a mechanism to obtain a pager object-cache object channel given a memory object. This mechanism is described in Section 4.l.

Operation	Description
flush_back	Remove data from the cache and send modified blocks to the pager.
deny_writes	Downgrade read-write blocks to read-only and return modified blocks to the pager.
write_back	Return modified blocks to the pager. Data is retained in the cache in the same mode as before the call.
delete_range	Remove data from the cache—no data is returned.

TABLE 2. Cache object operations

Operation	Description
zero_fill	Indicate to the VMM that a particular range of cache is zero-filled. The data blocks in the range are held by the VMM in read-write mode.
populate	Introduce data blocks into the cache.

TABLE 2. Cache object operations

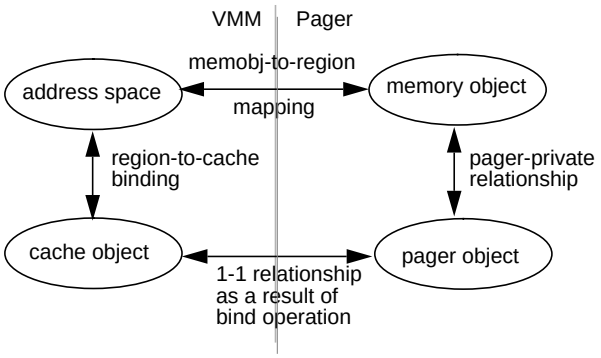

Figure 2. Relationship among basic VM objects

Figure 2 summarizes the relationship among the various objects. Note that the term “external pager” refers to the implementor of pager and memory objects. Strictly speaking, the VM architecture is defined in terms of the objects listed above and not the servers; implementations are free to use more than one server to provide these objects (see Section 5). As we will show, the implementations of the memory and pager objects can reside in different servers. As far as the VMM is concerned, it deals with pager and memory objects and it does not care where the implementations of these objects reside.

2.3 Maintaining Data Coherency

The task of maintaining data coherency between different VMMs that are caching a memory object is the responsibility of the pager for the memory object. The coherency protocol is not specified by the architecture—pagers are free to implement whatever coherency protocol they wish. The cache and pager object interfaces provide basic building blocks for constructing the coherency protocol.

3 Separating the Memory Object from the Pager Object

In the Spring operating system, the memory object *represents* memory and is separate from the pager object that actually *provides* the operations to page-in and page-out the memory. Unlike other external pager-based systems such as the MACH® operating system [5], the Spring memory object does not provide paging operations. Table 3 summaries the differences between a MACH memory object and a Spring memory object.

Separating the memory abstraction from the paging interface has the major advantage of giving the implementor of the memory object the power to place the implementation of the memory object in a separate server from the implementation of the pager object. For example, the Spring file system uses this separation to interpose a local attribute caching file system (CFS) in the local node, with the end result that all file attributes are cached by the CFS, file data is cached by the VMM, all reads and writes to the file go to the local CFS and use the data cached by the VMM, and yet all page-ins and page-outs go directly to the remote server where the data is stored on disk. We describe the CFS in Section 5.1. We also utilize the separation between memory and pager objects in our extensible file system architecture as will be described in Section 5.2.

	The MACH OS	**The Spring OS**
Memory Object	• memory mapped & encapsulates access rights • init/terminate ops • paging operations	• memory mapped & encapsulates access rights • bind operation • *no* paging operations
File Object	• can be same port as memory object • may provide file operations using paging ops	• inherits from memory object • provides file read/write operation • *no* paging ops

TABLE 3. Memory objects in the MACH and Spring OS

4 The Bind Protocol

When a VMM is asked to map a memory object into an address space, it needs to answer three questions:

- Is this memory object equivalent to a memory object that is already being cached at the VMM? If so, the new memory object can share the cached state of the equivalent object.
- What are the encapsulated access rights of the memory object? These are needed for access control to the shared cached state.
- What pager object should be used to get data?

One possible way of answering the first question is to associate a global identifier with each memory object. Each time the VMM is asked to map a memory object, it uses the global ID to see if the memory object is equivalent to a memory object that is already cached by the VMM. If the VMM finds an equivalent memory object, then it can use the associated pager object as a paging channel for the newly mapped memory object. If no such channel exists, the VMM can contact the memory object to establish a channel. This is basically the approach taken by the MACH virtual memory system. The advantage of this scheme is that it requires only one memory object initialization operation, regardless of how many times the memory object is mapped.

The problem with using global identifiers for memory objects is that it does not allow two distinct memory objects that encapsulate the same data to use the same cached memory. For example, in an object-oriented system such as the Spring operating system, it is common to have two or more distinct objects that encapsulate access to the same underlying state, each perhaps with different access rights. In a system that relies on global identifiers such as the MACH operating system, these distinct memory objects will have distinct global identifiers. Thus the VMM when presented with two distinct memory objects will not allow them to share the same cached state.

We made an early decision in the design of our VM system to allow different memory objects to encapsulate different access rights to the same memory. In particular, we wanted to allow different file objects to encapsulate different access rights to the same file while using the *same* physical memory to cache the contents of the file.

Instead of using global identifiers, the Spring operating system uses a special bind protocol that simultaneously allows the VMM to answer all three of the questions listed above. The bind protocol involves the implementor of memory objects, since the implementor can determine if two memory objects are equivalent (i.e., share the same underlying state) and can determine the encapsulated rights of the memory object. Thus, when the VMM is presented with a memory object to map, it invokes the *bind* operation on the memory object. The result of the bind operation is an object that allows the VMM to determine the encapsulated access rights for the memory object and the associated cache and pager objects. The implementor ensures when it is asked by the same VMM to bind two distinct yet equivalent memory objects, that the VMM will use the same cached data.

The decision to call the bind operation on each map request raises three issues:

1. Cost of the bind operation. Since the VMM has to call the memory object's bind operation on each map request, it is important to minimize the cost of this operation. By using the CFS (Section 5.1), the cost of the bind operation is exactly one local object invocation when the file is cached on the local machine.
2. Security. It is imperative that the VMM is not fooled by a malicious (or an incompetent) memory object implementor into using a cache-pager channel that belongs to a different client (Section 4.2).
3. Cache reclamation. It is important that the VMM is allowed to reclaim all resources associated with unused cache objects, while allowing bind operations to proceed in parallel (Section 4.4).

4.1 Protocol Description

Figure 3 shows the sequence of operations made during the bind operation. The step numbers correspond to the following list:

1. An application issues a request on an address space object that requires mapping a memory object to a region in the address space (or a request to make a copy of a memory object).
2. The VMM is presented with a memory object to map (or to make a copy from the object). It needs to associate the mapping with a local cache object. Therefore, it calls the *bind* operation on the memory object, requesting from the pager a cache object to use when accessing the mapped memory. The arguments of the bind request include the name of the VMM (*not* the object representing the VMM; see below), the length, and the access mode of the requested binding.
3. The pager that implements the memory object receives the bind request. It decides whether or not a cache object that caches the state of the memory object already exists at the requesting VMM.

   3a. If no cache object that caches the contents of the memory object exists at the VMM, a *create_cache* call is issued to the VMM.[1]

   3b. The VMM receives a *create_cache* call that includes a pager object as an input argument.

   3c. The VMM returns a cache object plus a list of *cache-rights* objects. A cache-rights object is a Spring object that represents the right to access a cache object with an encapsulated access right. It is used as a secure capability.
4. The pager returns from the bind call by pointing the VMM to an existing local cache object that caches the contents of the memory object. The "pointer" used by the VMM is a cache-rights object and not the cache object itself.

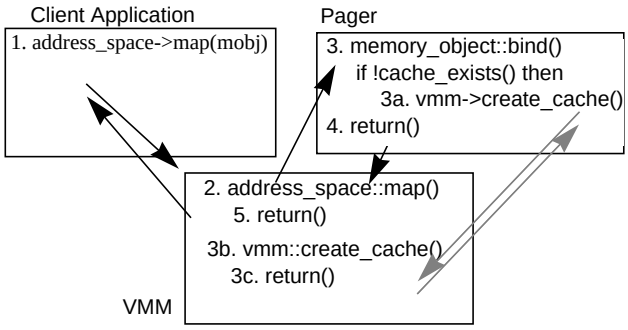

Figure 3. The bind protocol

(1) Client requests mobj to be mapped in an address space; (2) the map request is turned around into a *bind* on mobj. (3) If a cache that backs mobj does not exist at the calling VMM then (3a) a new cache is created at the VMM, etc. (4) The *bind* call returns pointing the VMM to a local cache object. (5) The VMM uses this cache object to back the requested address space region.

1. The pager first looks up the *vmm* object given the name passed in the original bind call if it does not have the VMM object cached already. The name lookup is made on some well-known name server in an authenticated manner.

5. The VMM uses the cache-rights object to find the corresponding cache object and completes the original client request by checking the requested access mode against the access mode encapsulated in the cache-rights object.

4.2 Cache-Rights Object

The cache-rights objects returned in Step 3c above are used as secure capabilities. A cache-rights object encapsulates an access right to a cache object and supports one operation: to obtain other cache-rights objects that encapsulate a *subset* of the encapsulated access rights. The VMM supports four possible access rights:

- read-only
- read-execute
- read-write
- read-write-execute

A cache creation request from the pager includes the maximum access rights of the cache being created. The VMM returns at least one cache-rights object that encapsulates the maximum requested access right. The holder of the cache-rights object may obtain weaker versions of the object by calling the *create_restricted_sibling* operation on the object. For example, a read-only cache-rights object can be obtained from a read-write cache-rights object, but a read-write-execute object cannot be obtained from a read-write cache-rights object. The rationale behind using the cache-rights object is explained in the next section.

4.3 Discussion

We argue in this section for the correctness of the protocol:

- When a VMM is given a memory object to map, it needs to find a corresponding cache object. The only entity it can request this information from is the memory object itself.
- However, the VMM must be sure that when it asks the memory object, "Give me a cache object that has your data," that the answer points to the correct cache object and not to some other cache object thus compromising security. The VMM does not trust pagers to be honest. The VMM trusts pagers only with the data they are supposed to manage.[2]
- Therefore, the VMM protects itself by requiring the pager to return as a result of the bind call a cache-rights object and not a forgeable identifier, since a forgeable identifier can give a malicious pager access to a cache that it should not control.[3] Similarly, the pager protects itself by returning a cache-rights object and not the actual cache object, since a cache-rights object is of use only to the implementor of the cache object and to nobody else. Note that a cache-rights object is a Spring object, and is not forgeable.
- When a pager receives a bind request from a VMM, it does not know for sure that the caller is really the VMM indicated in the call. Moreover, pagers do not trust the VMM except for handling the data of the memory object in question.
- Therefore, the VMM sends its name and not a VMM object in the bind request. It is up to the pager to use the name to look up an authenticated VMM object. Once an authenticated VMM object is obtained, the pager can issue a *create_cache* call knowing with certainty that it is invoking the right VMM. Note that pagers can look up an authenticated VMM object once and cache it for future *create_cache* calls.

If the system is structured such that the pager, the VMM, and the original client are on the same node and a cache object already exists at the VMM, an address space *map* request can be satisfied by issuing two local object invocations: the address space *map* call and memory object *bind* call. Our file system uses such an implementation as described in Section 5.1.

4.4 Cache Reclamation

Pagers may delay deleting unattached caches in the hope of reusing them later on. The VM system tries to retain as many unattached caches as it can. However, as with any resource, the system has to impose a limit on the maximum number of unattached caches. Therefore, the VMM has to reclaim some of these caches when the limit is reached.

---

2. If a pager does not even handle its own data correctly, then the only losers are those clients that depend on its memory objects. Looking up a memory object from a pager in a secure manner is the responsibility of these clients and not the responsibility of the VMM.

3. Alternatively, one could send an encrypted identifier. As with other parts of the Spring system, we made a conscious decision to avoid using encryption and instead used a Spring object. This way we hide encryption, if any, in the support the system provides for secure Spring objects and not in application code.

It is possible that while a pager is returning from a *bind* operation, the VMM may decide to reclaim the same cache referenced in the call. Since it is not acceptable to fail the *bind* call (and the corresponding address space *map* call), the bind protocol needs to be extended to recover from this race. Although this race is seldom encountered in practice, a recovery protocol is necessary nonetheless.[4]

The protocol extension is the following: When the *bind* call returns, the VMM checks to see if the cache-rights object points to a valid cache. If it does not, then instead of failing the *map* call, it invokes the *final_bind* operation on the memory object, passing in addition to the usual bind arguments a *bind-key* object which has no operations. Note that at this point, the VMM does not know whether it has hit a cache reclamation race or it is simply dealing with a malicious/incompetent pager.

When the pager receives the *final_bind* call, it is expected to call the VMM passing the bind-key object to the *create_cache_object_and_bind* or the *bind_cache* call. The pager calls the latter operation when it believes that it has a cache at the VMM. If the *bind_cache* call fails, the pager then calls the *create_cache_object_and_bind* operation.

When the VMM receives either call, it uses the passed bind-key object to identify the outstanding *final_bind* call and associates the new cache with that call. A successful execution of a *bind_cache* or a *create_cache_object_and_bind* guarantees that a cache is bound to a bind-key object, and that this cache will not be deleted until the bind-key object is deleted.

When the *final_bind* call returns to the VMM, it checks to see if a *create_cache_object_and_bind* or a bind_cache was executed successfully. If so, the original *map* request is satisfied, otherwise the VMM fails the *bind* and the corresponding *map* request.

4. In practice, the only time we observe this recovery protocol executing is when a machine reboots faster than a remote pager notices the quick reboot.

5 Examples

5.1 Caching File System

The Spring operating system provides a coherent distributed file system (DFS). When a client resolves a file name through the naming system, it receives a file object with the requested access rights. As with other Spring objects, file objects can be freely passed around the network. A DFS acting as an external pager handles bind requests from the local VMM and remote VMMs and is responsible for keeping the different caches consistent [3].

If a client obtains a file object that is implemented by a remote DFS, all file operations and binds on the file objects result in network RPCs to the remote DFS, and no caching of file attributes or file read/write operations is done. (Of course, the local VMM caches memory-mapped contents of the file if the file is mapped locally.)

To enable caching of file attributes and read/write operations, a CFS is introduced on each machine. The main function of the CFS is to interpose on remote files when they are passed to the local machine as described in [3, 6]. Once interposed on, all calls to remote files end up being intercepted by the local CFS (Figure 4).

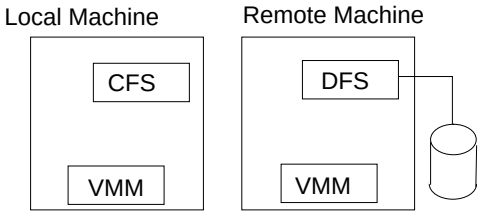

Figure 4. Using CFS to cache remote file objects

File objects exported by the remote DFS are cached by the local CFS. All files operations are serviced by the CFS, and all paging operations go to the remote DFS where the disk is located. File read/write operations are serviced from memory cached by the local VMM.

An important function of the CFS is to handle bind requests for remote files. When a remote file is mapped locally, the VMM invokes the bind operation on the file. Since the file is interposed on by the CFS, the CFS

receives the bind. The CFS proceeds by returning to the VMM a cache-pager channel to the *remote* DFS. Therefore, all page-ins and page-outs from the VMM go directly to the remote DFS.

The CFS also caches all file attributes from the remote file system and cooperates with the remote file system to keep them coherent using its own attribute coherency protocol. Finally, the CFS services read/write requests by mapping the file into its address space and reading/writing the data from/to its memory (thus utilizing the local VMM cache for caching the data).

The CFS utilizes the separation of the memory object from the pager object to ensure that all VMM page-in and page-out requests go directly to the pager object implemented by the remote file system, while at the same time handling all file (and memory object) operations locally.

Therefore, using the CFS, all file operations including bind requests are handled locally. However, the CFS is optional. If it is not running, remote files are not interposed on and all file operations go to the remote DFS. Reference [3] describes in detail the architecture and implementation of the file system.

5.2 Stacking File Systems

The Spring operating system provides an extensible file system architecture that allows for a file system to be composed (or stacked) on top of other file systems [7]. A file system normally acts as a pager by implementing pager and memory objects. It can also act as a *cache manager* (similar to a VMM) by implementing cache objects.

A goal of the extensible file system architecture is to allow new file systems to be implemented using other file systems, without necessarily re-implementing all the functionality provided by the existing file systems. The separation of the memory object from the pager object allows a file system to implement the memory object (the file) without implementing a corresponding pager object. A file system can relegate paging services to an underlying file system, by forwarding an incoming bind request to the underlying file.

We have found the separation of the memory object from the pager object very useful in implementing stackable files. For example, a file system that exports local files through some private protocol (e.g., AFS [8]) can handle remote bind requests itself, while forwarding local binds to the underlying file system. More details are available in [7].

6 Related Work

There are several systems that provide rich virtual memory subsystems that support the notion of external pagers [5, 9, 10]. In this section we concentrate on discussing the notion of separating the memory from the pager objects as it relates to the MACH operating system and the CHORUS® operating system. (See [1, 2, 6, 7] for other comparisons of the Spring operating system, the MACH operating system, and the CHORUS operating system.)

6.1 The MACH Operating System

The MACH operating system has a virtual memory system that supports an external pager interface [5]. Unlike the MACH operating system, the Spring operating system separates the memory object from the object used for paging operations (the pager object). In the MACH operating system, these two objects are one and the same although they provide different functionality: the first encapsulates access to a (logical) piece of memory, while the other is used to obtain the physical underlying memory.

The Spring operating system also differs from the MACH operating system in that the Spring operating system provides different views on the same memory. To achieve a similar effect in the MACH operating system one has to:

a. Have a copy of the data per memory object and force the pager to copy the data between the different memory objects even on the same machine, or

b. Develop a protocol based on a third trusted agent that sits between the system and the client (e.g., see [5], page 103), or

c. Modify the external pager interface, perhaps along the lines of our system.

As a final difference, file objects in the Spring operating system support read and write operations. As mentioned in [5], a holder of a MACH memory object may read and write its contents by using the paging operations provided by the object. To do so, however, requires the client to act effectively as a VM system, engaging the pager in the

memory object's paging and initialization/termination protocols. In practice, UNIX applications on MACH access files through an emulation library that maps the file in the process' address space [12].

Although one may argue that it is cheaper to access the file by mapping it rather than by reading/writing to it (which probably requires someone to map it somewhere anyway), in the Spring operating system, we wanted to retain the ability to issue read/write requests on the file object directly. The file interface in the Spring operating system inherits from the I/O interface, and any operation that expects an I/O stream may be passed a file. Therefore, clients that expect a stream will act on the file as a stream, issuing read and write operations on the file object directly without going through an emulation layer.

6.2 The CHORUS Operating System

The memory *segment* in the CHORUS operating system is similar to MACH's memory object. Segments are managed by *memory mappers* that provide operations to page-in and page-out the segment [9]. A segment may be memory mapped or explicitly read and written through CHORUS system calls [11]. Basically, all segment operations are sent to the mapper's port.

The CHORUS/MiX V.4 subsystem adds a local mapper per-node in addition to a global mapper [11]. These mappers cooperate with the *file managers* to provide consistent access to files in a distributed system. When a file is opened by the CHORUS process manager (PM), the file system contacts the global-mapper to construct a so-called *coherent capability* that is returned to the PM. Paging calls on this capability are then routed to the local mapper which in turn forwards the call to the remote global mapper if the data is not available locally.

The local mapper serves a different function from our CFS. CFS is used to intercept operations on remote files, and is not involved in paging operations, due to the separation of the memory and pager objects. The local mapper on the other hand is consulted on all paging operations. It is not clear from [11] how and if file attribute caching is done in CHORUS/MiX V.4.

7 Conclusions and Future Work

We have designed and implemented a virtual memory system for a general-purpose operating system that emphasizes object-oriented interfaces, security, and distribution. The VM system separates the memory abstraction from the interface that provides the actual data. The VM system provides an architecture for efficient sharing of memory in a secure manner, and for building distributed extensible file systems.

The notion of separating the memory abstraction from the paging interface is simple, but powerful. We have found the separation of the memory and pager objects to be very useful in the implementation of several pagers.

Although somewhat orthogonal to separating the memory object from the pager object, we also found that it is very useful to be able to encapsulate the access right in the memory object separately from the pager object. Finally, the ability to directly issue read and write operations on files (separate from paging operations) is important in an object-oriented system where files are also io streams.

All functionality described in this paper has been implemented and is part of the base Spring system. The system currently runs on several uniprocessor and multiprocessor SPARCstation™ models. All system servers including the kernel are multi-threaded and are written in C++.

We are continuing our work in virtual memory and the file system. We are currently implementing new file system layers as part of our extensible file system work, and evaluating and tuning the performance of the system.

References

[1] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-oriented Operating System." *Proceedings of Winter '93 USENIX Conference* (January 1993): 469-480.

[2] Khalidi, Yousef A. and Michael N. Nelson. "The Spring Virtual Memory System." *Sun Microsystems Laboratories Technical Report SMLI TR 93-09* (February 1993).

[3] Nelson, Michael N., Yousef A. Khalidi, and Peter W. Madany. "The Spring File System." *Sun Microsystems Laboratories Technical Report SMLI TR 93-10* (February 1993).

[4] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-Independent

Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computers* 37, no. 8 (August 1988): 896-908.

[5] Young, Michael Wayne. "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System." *Carnegie Mellon University Technical Report CMU-CS-89-202* (November 1989).

[6] Nelson, Michael N., Graham Hamilton, and Yousef A. Khalidi. "A Framework for Caching in an Object-Oriented System." *Sun Microsystems Laboratories Technical Report SMLI TR 93-19* (October 1993).

[7] Khalidi, Yousef A. and Michael N. Nelson. "Extensible File Systems in Spring." *14th Symposium on Operating System Principles (SOSP '93)* (December 1993).

[8] Satyanarayanan, S. "Scalable, Secure, and Highly Available Distributed File Access." *IEEE Computer* 23, no. 5 (May 1990): 9-21.

[9] Abrosimov, Vadim, Marc Rozier, and Marc Shapiro. "Generic Memory Management for Operating System Kernels." *12th Symposium on Operating Systems Principles (SOSP `89)* (December 1989): 123-136.

[10] Harty, Kieran, and David R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management." *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (September 1992): 187-197.

[11] Abrosimov, Vadim, Francois Armand, and Maria Inés Ortega. "A Distributed Consistency Server for the CHORUS System." *3rd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)* (March 1992): 129-148.

[12] Dean, R. W., and F. Armand. "Data Movement in Kernelized Systems." *Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures* (April 1992): 243-261.