Generic Virtual Memory Management for Operating System Kernels*

Vadim Abrossimov, Marc Rozier
Chorus systèmes
Marc Shapiro
INRIA

Abstract

We discuss the rationale and design of a Generic Memory management Interface, for a family of scalable operating systems. It consists of a general interface for managing virtual memory, independently of the underlying hardware architecture (e.g. paged versus segmented memory), and independently of the operating system kernel in which it is to be integrated. In particular, this interface provides abstractions for support of a single, consistent cache for both mapped objects and explicit I/O, and control of data caching in real memory. Data management policies are delegated to external managers.

A portable implementation of the Generic Memory management Interface for paged architectures, the Paged Virtual Memory manager, is detailed. The PVM uses the novel history object technique for efficient deferred copying. The GMI is used by the Chorus Nucleus, in particular to support a distributed version of Unix. Performance measurements compare favorably with other systems.

1 Introduction

Memory management services and implementations are generally highly dependent on the operating system. Different classes of operating systems use different classes of memory management services:

- In *real-time* executives, memory management services are primitive. Most real-time operating systems do not exploit the hardware MMU. They are just beginning to integrate the concept of protected address spaces.
- *General-purpose* operating systems, such as Unix[1], integrate virtual memory management services, allowing protected address spaces to co-exist on a limited hardware.
- Some *distributed* operating systems support distributed virtual memory schemes such as [8]. Until recently most were research projects (with the notable exception of the Apollo Domain [7]), some are now becoming products, e.g. Mach [13] and Chorus [15].

The Chorus[2] architecture is designed to support new generations of open, distributed, scalable operating systems. It allows the integration of various families of operating systems, ranging from small real-time systems to general-purpose operating systems, in a single distributed environment.

The Chorus architecture is based on a minimal real-time Nucleus that integrates distributed processing and communication at the lowest level. Chorus operating systems are built as sets of independent system servers, that rely on the basic, generic services provided by the Nucleus, i.e. thread scheduling, network transparent IPC, virtual memory management and real-time event handling.

*In: proceedings of the 12th ACM Symposium on Operating System Principles (SOSP '89), Litchfield Park, Arizona, December 3–6, 1989.

[1] Unix is a registered trademark of AT&T
[2] Chorus is a registered trademark of Chorus systèmes

The Chorus Nucleus itself can be scaled to exploit a wide range of hardware configurations, such as embedded boards, multi-processor and multi-computer configurations, networked workstations and dedicated servers.

Operating systems currently implemented on top of this Nucleus are, for instance, Chorus/MIX, a Unix System V compatible distributed real-time system [6, 2], and PCTE [10]. Work is currently in progress to implement object-oriented distributed systems [16, 9].

The design of the right memory management service was a delicate task, due to the multiple purposes of the Chorus Nucleus. The memory management service must be a *replaceable* unit, independent from the other Nucleus pieces. Therefore, we defined the "Generic Memory management Interface" (GMI). The GMI is suitable for various architectures (e.g. paged and/or segmented) and implementation schemes; it is scalable, and kernel-independent. We present in detail the architecture of the PVM, a demand-paged virtual memory implementation of the GMI. The PVM uses *history objects*, a novel technique for deferred copying. The PVM is hardware-independent.

The outline of the rest of this paper is the following. In section 2 we briefly present an overview of the memory management services, as seen by a user program. Section 3 describes the architecture: major abstractions, layering, and interface. In section 4, we focus on the PVM. We describe history objects, which we compare with the "shadow objects" of Mach. Section 5 describes the integration of the GMI in the Chorus Nucleus, and presents some encouraging performance measurements.

2 Basic services

A memory management subsystem must support execution of independent programs, and data transfer on their behalf.

It will provide separate address spaces (if the hardware gives adequate support), into which the code of a program is mapped, along with the data it accesses. Address spaces will be called "contexts" in the remainder of this paper.

It will provide efficient and versatile mechanisms for data transfer between contexts, and between secondary storage and a context. The mechanisms must adapt to various needs, such as Inter-Process Communication (IPC), file read/write or mapping, memory sharing between contexts, and context duplication.

Our memory management system considers the data of a context to be a set of non-overlapping *regions*, which form the valid portions of the context.

In this paper, we consider Memory Management as an independent component of the operating system kernel. It offers an architecture-independent Generic Memory management Interface, the GMI, to the other kernel components. Secondary storage objects, called *segments* in this paper, are assumed to be managed outside of the memory manager subsystem, by external servers, called *segment managers*. These servers manage the implementation of the segments, as well as protection and designation. They provide a simple segment access interface (described in section 3) to the Memory Management. A memory manager accesses this interface by upcalls across the GMI.

The "host" kernel for the Memory Management must provide a simple synchronization interface, to allow concurrent Memory Management operations.

3 Architecture

We will now define precisely the memory management abstractions, and describe the GMI. Please refer to Figure 1 for all of this section.

3.1 Memory management layers

The memory management architecture defines a generic, kernel-independent, architecture-independent memory management interface, the GMI.

Above the GMI is a kernel-dependent layer for system calls, IPC, and synchronization.

Underneath the GMI is a particular memory manager (MM) for some memory architecture; in this article we concentrate on the PVM, for demand-paged virtual memory. The PVM is designed and implemented independently of a particular addressing scheme or hardware memory management unit (MMU). The few dependencies to a particular MMU are insulated under a hardware-independent PVM interface.

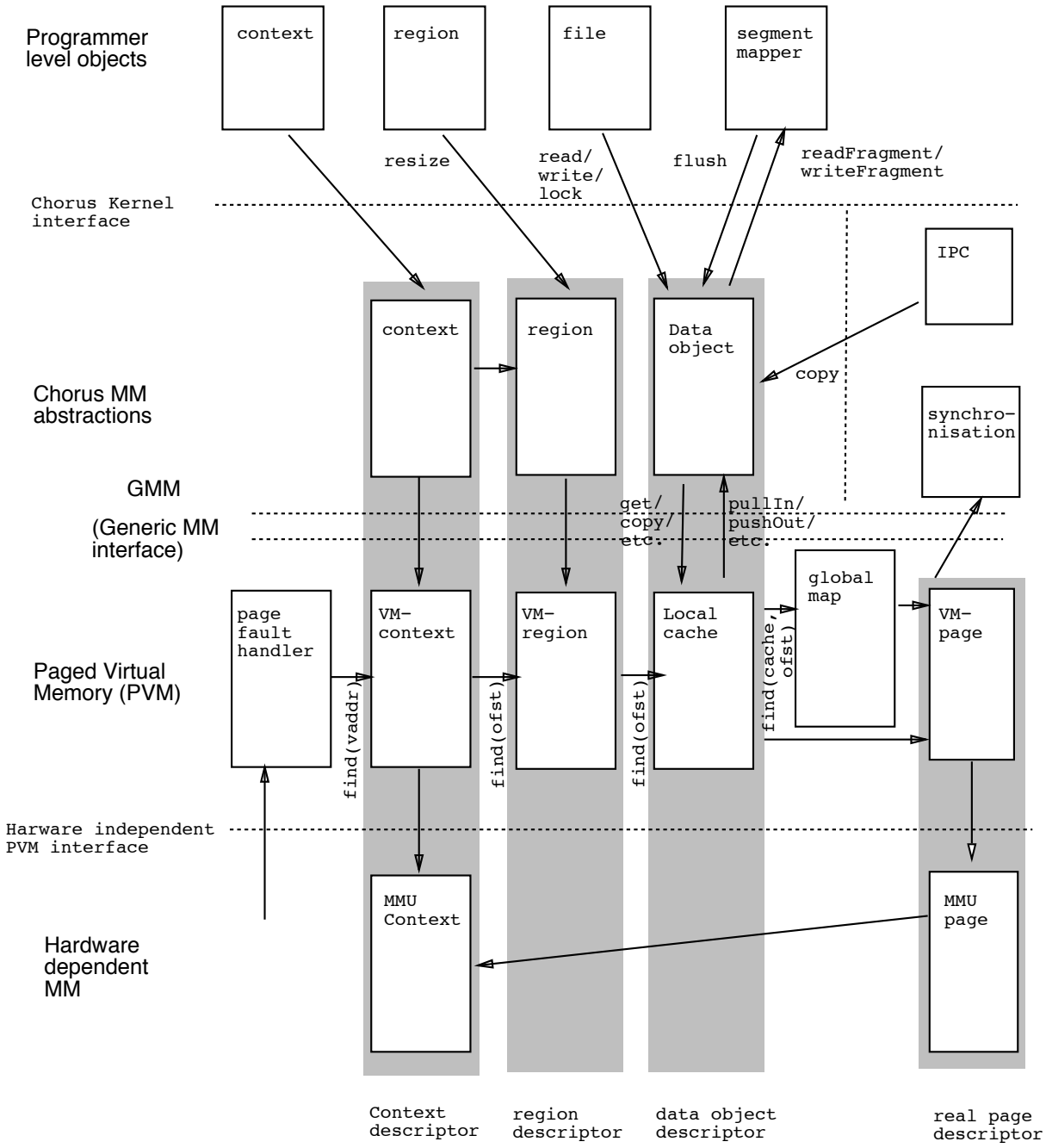

Figure 1: Memory Management Architecture

3.2 Memory management abstractions

In first approximation, a *context* is identical to a program's protected virtual address space. A context is sparsely populated with non-overlapping *regions*, separated by unallocated zones. A region is a contiguous portion of virtual address space. Some regions are *sparse*, i.e. their contents is mostly undefined.

A region is mapped to a *segment* through a *local cache*, an object which manages the real memory currently in use for a particular segment.

A region may map a whole segment, or may be a window into part of it. A protection (e.g. read/write/execute, user/system) is associated with each entire region. Different parts of a segment can be protected differently, by mapping each to a separate region.

In a Unix-like system with demand-paging, there are two potential conflicts between read/write and mapped access to segments. Firstly, the file buffers and the page buffers compete for real memory, which can lead to contention and a poor utilization of real memory. Secondly, if a segment can be both mapped and read/written, and if each access has its own cache, the two caches can become inconsistent; this is known as the dual caching problem [11]. The GMI solves these problems by offering a unified interface to segments: in addition to the mapped-memory access described above, the same cache can be accessed by explicit data transfer through copy (i.e. read/write) operations.

Concurrent access to a segment is allowed: a given segment may be mapped into any number of regions, allocated to any number of contexts; it can also, at the same time, be accessed by copy operations, again from any number of contexts.

The GMI defines the operations on regions, contexts, segments and local-caches. The segment is implemented above the GMI (see section 3), whereas the others are implemented below the GMI, as we will now describe.

3.3 Memory management interface

The following sections and tables describe the GMI. This is a faithful description of the real GMI, but some needless detail has been abstracted away. The procedures do not check for logical errors, such as an out-of-bounds offset, which are assumed to have been checked by the upper layers of the kernel. Other problems, such as resource exhaustion, may cause error returns; these are not indicated here.

cacheCreate (segment) → cache *bind a new cache to a segment*
cache . copy (offset, size, srcCache, srcOffset) *copy a fragment from another segment*
cache . move (offset, size, srcCache, srcOffset) *move a fragment from another segment*
cache . destroy () *destroy, flushing all modified portions back to segment*

Table 1: GMI: segment access.

All these primitives, except those of Table 3, are memory management procedures called by the upper layer of the kernel. Table 3 describes *upcalls* performed, by the memory management upon segment managers, to initiate the movement of data between a cache and its associated segment.

3.3.1 Copy data access

A segment is always accessed via its corresponding cache. Table 1 describes the cache operations pertaining to explicit data access.

The cacheCreate operation binds a segment to a newly-created (empty) cache. The cache can then be used in explicit data transfer operations (move and/or copy). It can also be used to create a mapping of the segment into some existing virtual address space, with the regionCreate operation. (Operations on regions are described in the next section.)

The copy operation copies data from a source cache (segment) to a destination cache (segment). Move is similar, except that the contents of the source becomes undefined; this allows the lower levels to implement it by changing the real-page-to-cache assignments, rather than by copying, whenever possible (i.e. if hardware and alignment allows it). Either operation may cause faults, which will cause it to block.

3.3.2 Mapped data access

Table 2 pertains to mapped data access; it describes the operations on contexts and regions.

The setProtection and lockInMemory operations associate hardware protection and in-memory pinning attributes to the whole region. In order to set different attributes on parts of a region, it can be split in two using the split operation. Splitting never occurs spontaneously; this allows the upper layers to keep track easily of the status of a region, and associate additional information with it.

After lockInMemory the data are pinned in real memory; furthermore, the underlying hardware MMU maps are guaranteed to remain fixed. This property is important for real-time kernels.

The getRegionList and getStatus operations allow to obtain useful information about the current state of a virtual address space.

contextCreate () → context *create an empty context (address space)* context . getRegionList () → regionList *list regions of context* context . switch () *set current user context* context . destroy () *destroy address space*
regionCreate (context, address, size, prot, cache, offset) → region *map a cache into context* region1 . split (offset) → region2 *cut a region in two* region . setProtection (prot) *change hardware protections* region . lockInMemory () *ensure access to region without faults* region . unlock () *faults may occur during access to region* region . status () *return address, size, protection, cache, etc.* region . destroy () *unmap corresponding cache from context*

Table 2: GMI: address space management.

3.3.3 Cache management

This section describes the interface for cache management (as opposed to cache access).

The data management policy (e.g. page-in and page-out decisions) is performed by the memory manager

segment . pullIn (offset, size, accessMode) *read in data from segment* segment . getWriteAccess (offset, size) *request write access* segment . pushOut (offset, size) *write data to segment* segmentCreate (cache) → segment *create segment*

Table 3: GMI to segment manager upcall interface.

(MM) implemented underneath the GMI. The MM performs the requests described in Table 3, as upcalls to the appropriate segments. Conversely, the cache management downcalls of Table 4 are available to segment managers. The MM may unilaterally decide to cache a fragment of data. When it needs data, it calls the pullIn operation of the corresponding segment. The segment implementation provides the data using fillUp operation. Cached data carries the access rights defined by the accessMode argument to pullIn; when a write access to read-only cached data occurs, the MM invokes getWriteAccess, to request write access.

When, at the time of a cache synchronization, flush, or destruction, the MM needs to save a fragment of cached data, it calls the pushOut operation on the corresponding segment. The MM gets the data from the cache using copyBack or moveBack.

While a pullIn or a pushOut operation is in progress, any concurrent access to the fragment is suspended, until the operation terminates. For that reason the cache access operations copy and move of Table 1 are different from the operations fillUp, copyBack and moveBack described here (Table 4): the former may cause faults, whereas the latter are used to resolve faults.

The MM sometimes creates caches unilaterally; see for instance history objects in section 4.2. With the segmentCreate upcall, the MM may declares such a cache to the upper layer, so that it can be swapped out.

A segment server may need to control some aspects of caching. For instance, to implement distributed coherent virtual memory [8], it needs to flush and/or lock the cache at times. The GMI provides operations flush, sync, invalidate and setProtection to control the cache state. Sync and flush operation may cause pushOuts;

cache . fillUp (offset, size, srcCache, srcOffset) *fill a cache fragment with data*
cache . copyBack (offset, size, dstCache, dstOffset) *copy a cache fragment to be written back*
cache . moveBack (offset, size, dstCache, dstOffset) *move a cache fragment to be written back*
cache . sync (offset, size) *write all modified portions of a cache fragment back to segment*
cache . invalidate (offset, size) *invalidate cache fragment*
cache . flush (offset, size) *synchronize and invalidate fragment*
cache . setProtection (offset, size, prot) *set hardware protection of fragment*
cache . lockInMemory (offset, size) *pin fragment in real memory*
cache . unlock (offset, size) *permit a cache fragment to be flushed*

Table 4: GMI: cache management.

lockInMemory may cause pullIns.

4 The PVM: a demand-paged implementation

A portable implementation of the GMI for paged architectures has been developed in the Chorus Nucleus. It is referred to as the PVM (Paged Virtual memory Manager) and supports a number of hardware memory management architectures. It is characterized by:

- Support for large, sparse segments and large virtual address spaces,
- Efficient deferred copy (copy-on-write [3] and copy-on-reference),
- Easy and efficient portability to different paged memory management units (MMU's).

The PVM is layered into a hardware-independent layer (the PVM proper) and a (much smaller) hardware-dependent one, separated by a hardware-independent interface.

Techniques used to support large segments and address spaces are comparable to those of Mach. However, our approach to copy optimization is quite different. Two different techniques are used in order to allow optimization in different cases:

- *history objects* to defer the copy of large data, such as a big data segment for a Unix process.
- a per-virtual-page technique to copy relatively small amounts of data (e.g. an IPC message).

In section 4.1 we briefly describe the technique used for address space and cache implementation, independently of deferred-copy issues. Sections 4.2 and 4.3 describe the rationale and implementation of the history object and per-virtual-page techniques, respectively.

4.1 Large segments and address spaces

The key to the efficient support of large segments and virtual address spaces is that the size of the data structures should not depend on the size of those segments or address spaces.. The size of the management structures should depend only on the amount of physical memory, and possibly on some configuration parameters, such as the maximum number of contexts or caches.

4.1.1 Data structures

The basic memory management objects (see Figures 1 and 2) are the following.

There is a global list of all the context descriptors on the host.

There is a context descriptor per context, which refers to descriptors of the (doubly-linked) list of regions it contains, sorted by start address.

There is a region descriptor per region. Each region descriptor holds the region start address, size and access rights, and a pointer to the cache descriptor for the segment that the region maps, and its start offset in that segment. Two different regions may refer to the same cache descriptor.

A cache descriptor holds an identifier of its data segment. It also holds the (doubly-linked) list of its currently-cached real page descriptors. A page in that list may be replaced by a *synchronization page stub* (defined below).

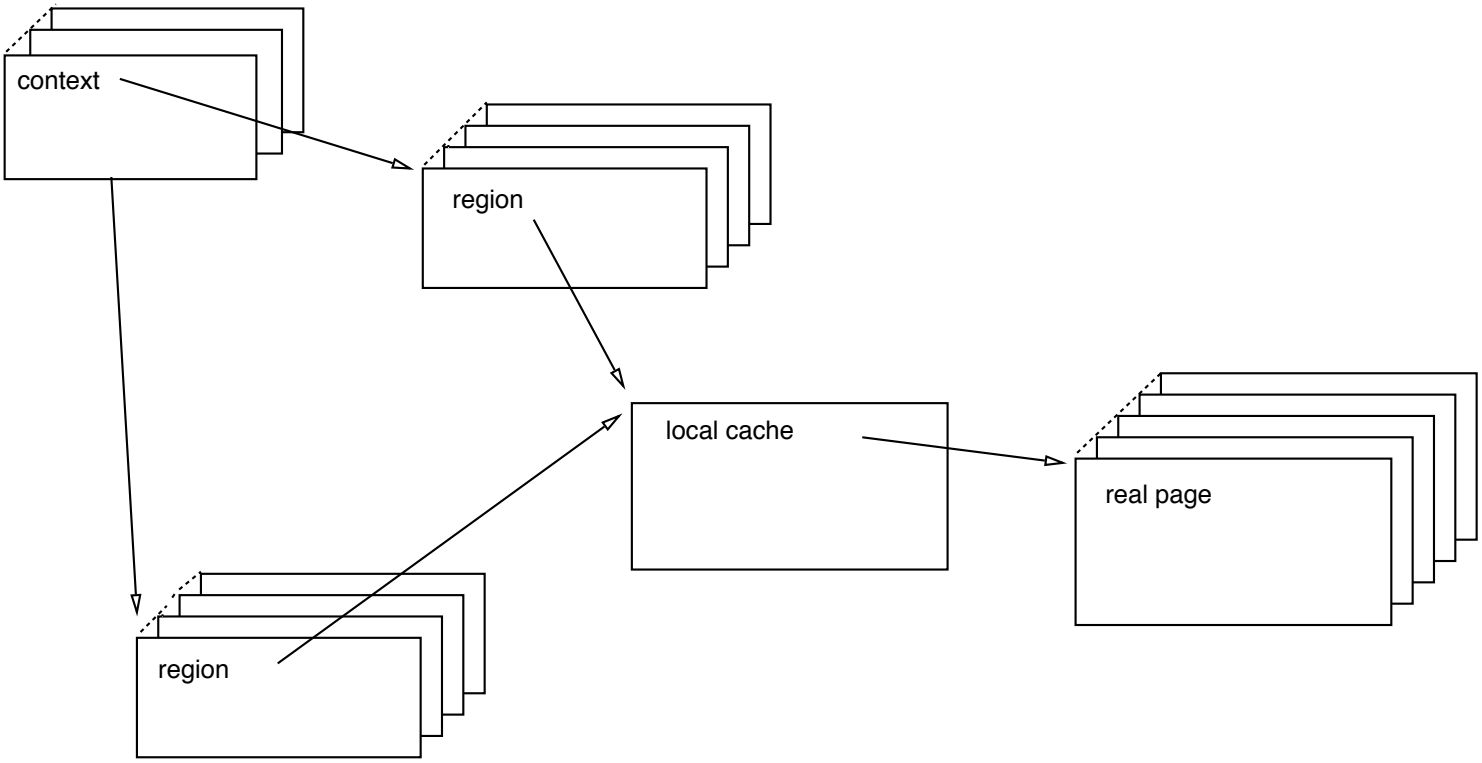

Figure 2: PVM data structures

A real page descriptor holds a back pointer to the cache descriptor, and the page's offset in the segment.

Furthermore, the PVM maintains a single *global map*, hashing real page descriptors by the page's cache, and its offset in the segment. The global map is used to find real pages efficiently.

4.1.2 Handling a page fault

When a page fault occurs, the hardware page fault descriptor holds the virtual address of the fault. Knowing the currently active context, the PVM searches in its list of region descriptors for the region containing the fault address. If the region is not found, the PVM raises the "segmentation fault" exception.

Otherwise, using the fault address, the region start address in the context, and the region start offset in the segment, the PVM computes the fault offset in the segment. The PVM then uses this offset and the identifier of the cache descriptor, to look up the page in the global map. If it is found, the page is already in physical memory and the page fault can be recovered immediately.

Otherwise, the data are not in physical memory and the pullIn operation shall be invoked on the segment. Before calling pullIn, the PVM places a synchronization page stub in the global map for that page. This will cause any future access to the virtual page to sleep, as long as it is in transit. When pullIn returns, the synchronization page stub is removed and replaced with the received page descriptor.

4.2 History objects

We use the novel *history object* technique to defer copies of large amounts of data. This technique can be used to implement both copy-on-write and copy-on-reference policies.

First, we describe the case when a new segment is created as a copy of a fragment of another. Then we discuss the case of a copy between existing segments. Finally we compare history objects to the "shadow objects" of Mach.

In the following description, for simplicity reasons we consider that all relevant pages are in memory. Considering swapped-out pages presents no extra difficulty but would obscure the presentation.

4.2.1 History trees

As copies take place between segments, we construct trees of their cache objects, as shown in Figure 3. A tree is rooted at the source of a copy; successive copies add new leaves. The following shape invariant holds: it is a binary tree, and each source of a copy operation has a single immediate descendant, called its history object.

Each cache contains the current version of its own pages. Pages not present in some cache (cache misses) are found by looking upwards (towards the root) in the tree. For this purpose, each node holds a pointer to its parent.

As pages are modified in the source of a copy, their original version is placed in its history object. Therefore each source holds a pointer to its history.

We will now explain by example how the tree gets constructed and used.

4.2.2 The simple case

Initially, a new segment cpy1 is created as a copy of a fragment of source segment src (see Figure 3.a). The cache for src will be at the root of the tree; cpy1 is its single descendant, and also its history object. When the data in cpy1 is accessed, any page not in cpy1 will be found, searching upwards in the tree, in src.

Copy-on-write is implemented as follows. When the copy cpy1 is created, all the pages of (the corresponding fragment of) the source src are made read-only. When a write violation occurs in the copy, a new (unprotected) page frame is allocated for the copy, and its value is copied from the corresponding src page. When a write violation occurs in the source, two cases are possible. If the history object (i.e. cpy1, in this case) already has its own version of the page, it suffices to make the page writable. Otherwise, an unprotected page frame is allocated in the history object, the data copied into it, and the source page is made writable.

A copy-on-reference scheme is implemented in a similar fashion. Immediately after cpy1 is created, access to any of its pages will fault; at that point a copy is allocated in cpy1 as above. Similarly, a write violation in src will cause (a copy of) the original version of the page to be placed in its history, i.e. in cpy1.

When the copy segment is deleted, its cache may simply be discarded. This is the normal case in Unix: the

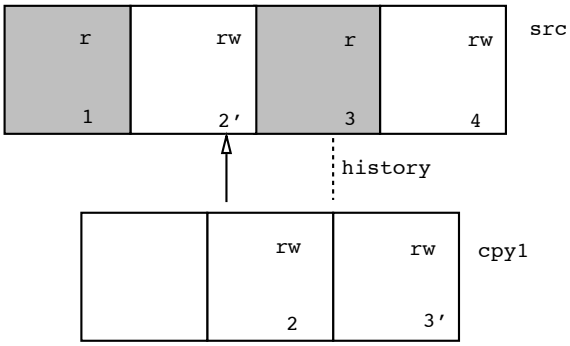

Figure 3a: cpy1 is a copy-on-write fragment 1-3 of src Page 2 has been updated in src, page 3 has been updated in cpy1. A cache miss on page 1 in cpy1 is resolved by looking it up in src. 2 stands for the original value of 2; 2′ stands for the new value.

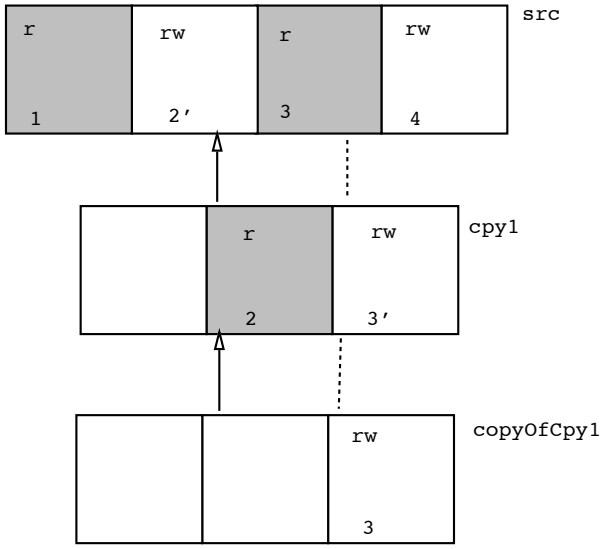

Figure 3b: src fragment 1-3 is copied-on-write into cpy1. Page 2 of src is modified. Then cpy1 is copied -on-write to copyOfCpy1. Page 3 of cpy1 is modified bothe src and copyOfCpy1 get a page frame with the original value. Page 1 of both copies is read from src. Page 2 of copyOfCpy1 is read from cpy1.

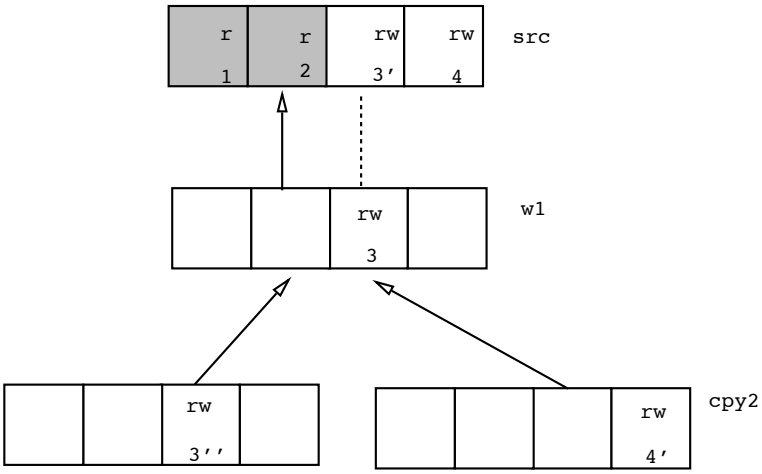

Figure 3c: src has been copied twice, into cpy1 and cpy2. A history object w1 has been created and inserted into the tree. The following pages have been modified: page 3 of src, page 3 of cpy1, and page 4 of cpy2.

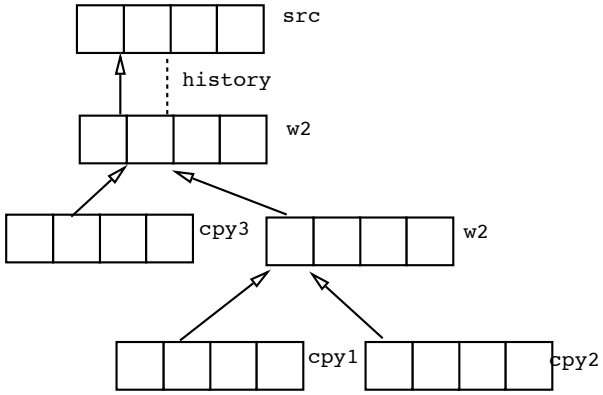

Figure 3d: src is copied-on-write three times. Two working history objects are created.

Figure 3: History objects for copy-on-write

source is the data segment of a process which forks; the copy is the child process's data. When the child exits, its data are deleted.

In the case where the source is deleted first (the parent process exits while the child continues), remaining unmodified source data must be kept until the copy is deleted (see section 4.2.5).

4.2.3 Successive copies

Suppose the cpy1 object in the previous example becomes in turn the source of a copy to segment copyOfCpy1, as in Figure 3.b. In Unix this occurs when a child process forks. The same tree construction algorithm applies: the tree is extended downwards. All the pages of (the corresponding fragment of) cpy1 are made read-only. The history object of cpy1 is copyOfCpy1. A cache miss in copyOfCpy1 goes to cpy1, and then possibly to src. The write-violation algorithm is the same as above (with the appropriate shift of roles, the source being cpy1, and the target and history being copyOfCpy1).

A small complication does arise in this case. When a write violation occurs in cpy1, a copy of the page is taken from src, but copyOfCpy1 must also get its own copy, since at the time of creation of copyOfCpy1, its value was logically taken from src.

Now, suppose src is again the source of a second copy, to cpy2 (see Figure 3.c); in Unix this occurs for instance when creating a pipeline, or with daemons. Then an intermediate "working" cache and segment w1 must be created to preserve the shape invariant. w1 is inserted between src and cpy1. w1 is the history object of src and the parent of both cpy1 and cpy2. A cache miss in cpy1 may be resolved either in w1 or in src; similarly for cpy2. At the time of the second copy, the corresponding pages of src must again be set read-only. The write violation algorithm above is unchanged (with the appropriate name substitutions: src is the source, cpy1 or cpy2 is the copy, and w1 is the history object).

If src is the source of a third copy, as in Figure 3.d, then again a working cache w2 is inserted in the tree to preserve its shape, and the src pages protected again.

4.2.4 Copying into an existing segment

Suppose we wish to copy a large amount of data into an existing segment. If the destination has been already initialized from another one, it already has a parent. Therefore, applying the above technique to the new copy requires a generalization, so that individual fragments may have different, arbitrary, parents.

To allow this, the "parent" attribute of a cache descriptor is in fact a list of parent descriptors. Each such descriptor holds the start offset and size of a fragment, and a pointer to the parent local-cache descriptor. The list is sorted by this offset.

4.2.5 Comparison with shadow objects

The history objects technique was inspired by the Mach's *shadow objects* [13]. When Mach initializes a cache (which they call a memory object) as a copy of an other, the source is set read-only, and two new memory objects, the shadow objects, are created. The shadows are to keep the pages modified by the source and copy objects respectively; the original pages remain in the source object.

If successive copies occur, a chain of shadows may build up. The current state of the source object is dispersed across the original object and its shadows; similarly for the copy. This causes some difficulties. For instance:

1. When a Unix process forks, the child's data segment is a copy of the parent's. After the fork, data modified by the parent is held by its shadow, even after the child exits. To prevent the creation of long chains of shadow objects, when the parent forks repeatedly (as do Unix shells), the shadow must be merged with the source after the child exits. This garbage collection is a major complication of the Mach algorithm [12].

2. The actual reference of a particular cache (i.e. the starting point for a cache look-up) changes dynamically as it is copied.

Our data structures are inverted with respect to the Mach structures. By construction of the history tree, the second problem does not occur. The history object technique eliminates the first problem for the source cache. The destination cache is a problem (i.e. chains of inactive history objects build up, which should be merged), only if a process forks and then exits, while its child continues, forks and exits, and so on. This kind of behavior is exceptional in Unix applications.

4.3 Per-virtual-page copy-on-write

When a copy of a relatively small fragment is required, a different optimization is applied: in this case, we use a per-virtual-page technique. Sprite [12] and SunOS 4.0 [5] defer copies on a per-virtual-page basis only.

For each page of the source fragment present in real memory, the PVM protects the page read-only. For all pages of the destination, it puts a *copy-on-write page stub* in the global map. The stub allows to find the corresponding source page: if the latter is in real memory, the stub contains a pointer to the source page descriptor; otherwise, it contains a pointer to the source local-cache descriptor and its offset within the source segment. All the stubs for some source page are threaded together on a list attached to its page descriptor. In this way, the source page is accessible, for reads, through any cache to which it was copied.

When a write violation occurs on a copy-on-write page stub, a new page frame is allocated with a copy of the source page, and inserted in the global map in replacement of the stub.

5 Application and experience

An operating system kernel integrating a GMI implementation must provide a *segment manager* and a set of basic synchronization mechanisms. That kernel uses the GMI to manage the address spaces according to its own memory management model.

We discuss, in section 5.1, some policies adopted by the Chorus Nucleus in its use of the GMI to implement the Chorus memory management interface [1], and the Chorus/MIX implementation of Unix. Sections 5.2 and 5.3 discuss the current status and some performance results, respectively.

5.1 The Chorus Nucleus and the GMI

This section starts with a brief description of Chorus Nucleus abstractions. In sections 5.1.2 and 5.1.3, we describe the functionality and implementation of the *segment manager*, the Nucleus interface between mappers and a GMI implementation. Section 5.1.4 describes the implementation of some Nucleus operations, based on the GMI and the segment manager. Section 5.1.5 describes the Chorus/MIX memory management. Finally, we discuss the relationship between memory management and IPC message passing.

5.1.1 Background

The physical support for a Chorus system [15] is composed of a set of *sites*, interconnected by a communications *network*. There is one Nucleus per site. A given site can support many simultaneous *actors*, i.e. address spaces, each supporting the execution of many parallel *threads*. Each actor normally has its own protected address space, but different actors may also use the same address space if necessary.

The Nucleus offers an IPC (Inter-Process Communication) message communication mechanism, allowing threads to communicate with each other (including across actor boundaries). Messages are not addressed directly to threads, but to intermediate entities called *ports*. A port is an address to which messages can be sent, and a queue holding the messages received but not yet consumed.

Nucleus memory management considers the text and data of an actor to be a set of non-overlapping *regions*, which form the valid portions of the actor address space. These regions are generally mapped to secondary storage objects, called *segments*.

A segment is implemented by an independent actor, its *mapper*, generally on secondary storage. Segments are designated by *sparse capabilities* (similar to Amoeba's [17]), containing the mapper's port name and a key. The key is opaque data of the mapper, allowing it to manage and protect segment access. A mapper exports a standard read/write interface, invoked using the IPC mechanisms. Some mappers are known to the Nucleus as *defaults*; these export an additional interface for the allocation of *temporary* segments.

5.1.2 Segment manager

The segment manager maps each segment used on the site to a GMI local-cache. Given a segment capability, the segment manager either finds the corresponding local-cache if it exists, or assigns one. A local-cache may be discarded (see section 5.1.3) when the segment is no longer in use on the site.

The segment manager associates a local-cache in use with a *local-cache capability*, containing the segment manager port name, a reference to the local-cache, and

some protection information. The cache control operations of Table 4 can be invoked by sending an IPC request to the segment manager, containing the appropriate capability. The segment manager, acting as cache server, transforms such a request into the corresponding GMI operation.

Similarly, the segment manager transforms a GMI upcall (of Table 3) into IPC upcalls to the corresponding segment mapper. For instance, when the memory manager calls pullIn, the segment manager sends an IPC read request, to the appropriate segment mapper port (taken from the segment capability). The request contains the segment capability and the local-cache capability, and the start offset, size, and access type of the required data. The mapper replies with a message containing the required data (transported as explained in section 5.1.6). Mappers may use the local-cache capability parameter to implement distributed consistency maintenance protocols above the different local-caches.

Finally, the segment manager may allocate a temporary local-cache. The segment manager waits for the first pushOut upcall for such a temporary cache to allocate it a "swap" temporary segment with a default mapper.

5.1.3 Segment caching

When some segment is no longer in use, the corresponding GMI cache could be discarded. Instead, the segment manager keeps such an unreferenced cache as long as possible, i.e. as long as there is enough free physical memory, and enough space in the sagment manager tables. When a program requests the use of a permanent segment, the manager first checks if there is a cache already kept for it. This segment caching strategy has a very significant impact on the performance of program loading (Unix exec) when the same programs are loaded frequently, such as occurs during a large make.

5.1.4 Nucleus memory management

The Nucleus interface contains high-level memory management operations, combining the functionality of a few GMI operations. We will describe a few examples of operations.

The Chorus rgnAllocate operation allocates a new memory region within an actor. To implement it, the the segment manager creates a temporary local-cache, which it maps into the actor using regionCreate the GMI operation.

Another operation, rgnMap, maps an existing segment into an actor. For this operation, the segment manager first finds (or creates) a corresponding GMI local-cache, and then maps it, using the regionCreate GMI operation.

The Chorus rgnInit creates a new region in an actor as a copy of an given existing segment. The segment manager creates a temporary local-cache, finds (or creates) the cache corresponding to the source segment, invokes cache.copy to initialize the new cache contents, and finally maps it, using regionCreate.

The rgnMapFromActor and rgnInitFromActor operations are similar to rgnMap and rgnInit, except that the source segment is designated by an address within an actor. These operations find the source local-cache using the context.findRegion and region.status GMI operations.

5.1.5 Chorus/MIX memory management

Chorus/MIX [6, 2] is a System V compatible Unix implementation in Chorus. Many of the functionalities of a standard Unix kernel are implemented by an actor, the process manager, which maps Unix process semantics onto the Chorus Nucleus objects. A standard Unix process is implemented as a Chorus actor hosting a single thread.

The Unix exec invokes the Chorus rgnMap operation to map the text segment of the process, rgnInit for its data segment, and rgnAllocate for the stack. A Unix fork uses rgnMapFromActor to share the text segment between the parent and child processes. It invokes rgnInitFromActor to create the child's data and stack areas as copies of the parent's.

5.1.6 IPC and memory management

IPC messages serve to transport data, both for users and for the system. Therefore we decouple IPC from memory management, in that IPC never has the side effect of creating, destroying, or changing the size of any region. In this sense, our concepts are more similar to the V-System's view [4] than to Mach [18]. However, IPC uses the per-page deferred copy, and the move semantics (see section 3.3.1), to optimize message transfers.

Messages are of limited size (64 Kbytes in the current implementation). They are not suitable for transferring large and/or sparse data. To transfer large or sparse data, users should call the memory management operations, and not IPC.

The kernel has a single fixed-sized transit segment, mapped in the kernel address space, made of 64 Kbyte slots. An IPC send is implemented as a cache.copy between the user-space segment and a transit slot, if the segment is large enough, otherwise as a bcopy. A receive is implemented by cache.move or bcopy.

5.2 Current status

We have made several different implementations of the GMI in the Chorus Nucleus:

- The PVM, described in this paper, suitable for general-purpose operating systems on paged hardware architectures.
- A minimal implementation, suited for embedded real-time systems and small hardware configurations.
- A simulation implementation that uses a Unix process as a virtual machine. This implementation is integrated into the Chorus Nucleus Simulator.[3]

Implementations of GMI for segmented (iAPX 286) and paged-segmented (iAPX 386) architectures are under development.

The MM implementation is the only difference between these Nucleus versions. All the other Nucleus components, which access memory management facilities via the GMI, are unaffected.

The Nucleus and the PVM are written in C++, and have been ported to various hardware: Sun 3, Bull DPX 1000 (a MC68020 workstation with a Motorola PMMU), Telmat T3000 (a MC68020-based multi-processor with a custom MMU), various MC68030 boards and AT/386 PC's. Work is in progress on several RISC architectures: SPARC (4Q89), MC88000 (4Q89) and ARM-3 (1Q90) processor based machines.

[3]The Chorus Nucleus Simulator is a Nucleus, implemented as a process on Unix systems. It is used as a development tool: it allows machine-independent kernel evolutions to be developed and validated comfortably. In addition, it is a practical teaching aid, and allows Chorus users to develop applications while the Nucleus is not yet ported on their hardware architecture.

On the memory management point of view, these different ports require only the rewriting of the (small) machine-dependent part of the PVM. On average, it takes about one man×month of work to port to a new MMU. Table 5 shows the sizes of the various components. The number of lines of code includes header files and comments. It does not include per-virtual-page deferred copy, which is not fully operational at the time of this writing. The Nucleus part includes the system call interface.

Machine Independent Part			
Component	C++ (lines)	assembler (lines)	object (bytes)
Nucleus MM part	1820	0	7.8 Kb
PVM: Machine-Independent	1980	0	7.5 Kb
Total	3700	0	15.3 Kb

MMU Dependent Part			
Component	C++ (lines)	assembler (lines)	object (bytes)
PVM: Machine-Dependent on Sun	790	150	3.2 Kb
PVM: Machine-Dependent on PMMU	1120	30	4.0 Kb
PVM: Machine-Dependent on iAPX 386	980	200	3.8 Kb

Table 5: Chorus Memory Management Components Sizes.

5.3 Performance

Two benchmark programs illustrate the performance of the Chorus virtual memory management based on the Paged Virtual Memory manager. These measure:

- The cost of allocating large, sparse regions,
- The overhead of deferred copy based on history trees, and
- The overhead of a real page copy, after delaying it.

For the purpose of comparison, those benchmark programs have also been run on Mach/4.3 operating system.

Chorus: zero-filled memory allocation				
region	actual allocation of real memory			
	0 Kb	8 Kb	256 Kb	1024 Kb
	0 pages	1 page	32 pages	128 pages
8 Kb	0.350 ms	1.50 ms	–	–
256 Kb	0.352 ms	1.60 ms	36.6 ms	–
1024 Kb	0.390 ms	1.63 ms	37.7 ms	145.9 ms

Mach: zero-filled memory allocation				
region size	actual allocation of real memory			
	0 Kb	8 Kb	256 Kb	1024 Kb
	0 pages	1 page	32 pages	128 pages
8 Kb	1.57 ms	3.12 ms	–	–
256 Kb	1.81 ms	3.19 ms	46.8 ms	–
1024 Kb	1.89 ms	3.26 ms	47.0 ms	180.8 ms

Table 6: Performance for zero-filled memory allocation.

Chorus: copy-on-write				
region size	Actual amount of data copied			
	0 Kb	8 Kb	256 Kb	1024 Kb
	0 pages	1 page	32 pages	128 pages
8 Kb	0.4 ms	2.10 ms	–	–
256 Kb	0.7 ms	2.47 ms	55.7 ms	–
1024 Kb	2.4 ms	4.2 ms	57.2 ms	221.9 ms

Mach: copy-on-write				
region size	Actual amount of data copied			
	0 Kb	8 Kb	256 Kb	1024 Kb
	0 pages	1 page	32 pages	128 pages
8 Kb	2.7 ms	4.82 ms	–	–
256 Kb	2.9 ms	5.12 ms	66.4 ms	–
1024 Kb	3.08 ms	5.18 ms	67 ms	256.41 ms

Table 7: Performance of copy-on-write.

The measurements presented below were made on a SUN-3/60 workstation with 8 megabytes of memory, 8-Kbyte pages, a MC68020 CPU running at 20MHZ, i.e. about 3 MIPS of processing power.

A copy (Unix bcopy) of 8 Kbytes in real memory, implemented in assembler, takes 1.4 ms. Filling 8 K bytes of real memory with zeroes (bzero) takes 0.87 ms.

5.3.1 Benchmarks

The first benchmark program creates a region, accesses some of the data within the region in order to demand allocation of filled-zero memory and, finally, deallocates the region. The following tables give the results of this benchmark on Chorus and Mach. For each region size, the table 6 shows the time elapsed for creating the region, allocating and deallocating some real memory, and destroying the region, averaged over some large number of iterations.

The second program creates a region, which is entirely allocated in real memory. It then copies it, and modifies some of the data within the source region (in order to force a real copy). The table 7 gives the results of this measurement. The source region is created and allocated before starting the measurement. For each region size, the table shows the time elapsed for creating the copy region, forcing a copy of some amount of data, and deallocating and destroying the copy region.

5.3.2 Discussion

The above figures show that the strong structure of our design does not preclude an efficient implementation. On the contrary: the simplicity of our machine-dependent part allows fine optimization with a minimal effort.

In Chorus, the cost of creating and destroying a region is practically independent of its size: the difference between creating a 1-page region and a 128-page region is only 10%. In fact, the region creation is totally independent of the region size, but its destruction requires the invalidation of the corresponding portion of the virtual address space. This is consistent with our initial goals.

The structural management overhead of a simple deferred copy initialization is of the order of 0.03 ms for the history tree (i.e. 10% of a simple region creation cost), plus 0.02 ms per page frame allocated in the initial region before the copy. The overhead per page is the cost of the page protection, calculated as the cost of a creation/copy of 128 pages region, minus the cost of a creation/copy of a one page region, divided by the number of additional pages, i.e. $(2.4\ ms - 0.4\ ms)/127$. The overhead of tree manage-

ment is calculated as the cost of a 1-page region creation/copy, minus the cost of creating and allocating 0 pages in a 1-page region, minus the per-page overhead, i.e. $0.4\ ms - 0.35\ ms - 0.02\ ms = 0.03\ ms$.

The overhead of copy-on-write (including the protection violation handling, page lookup in the history tree, new page allocation and mapping) is 0.31 ms per page. The formula used here is the cost of doing a deferred copy and a real copy of a region, minus the cost of a deferred copy of the same size region with no real copy, divided by the size of the region, minus the cost of copying a real page, i.e. $(221.9\ ms - 2.4\ ms)/128 - 1.4\ ms$.

The overhead of the history tree using may be deduced by comparing the last result with the cost of a simple on-demand page allocation, which is 0.27 ms. Here again, the overhead is of the order of 10%. The simple on-demand page allocation cost is calculated as the cost of creating (and deleting) and zero-filling a 128-page region, minus the cost of creating/deleting the same-sized region with no data allocation, divided by 128, minus the cost of filling a real page with zeroes, i.e. $(145.9\ ms - 0.39\ ms)/128 - 0.87\ ms$.

6 Conclusion

The multiple purposes of the Chorus kernel led to design its memory management as a truly independent, replaceable part. It provides a generic, architecture-independent, interface to the other components.

We identified generic memory management abstractions, matching the needs of different kinds of operation systems. They are independent of the particularities of different hardware architectures, while still allowing efficient implementations.

In this paper, we discussed in some detail one (hardware-independent) implementation of this generic interface, suited for state-of-the art demand-paged virtual memory. Our encouraging performance figures show the validity of our approach.

Acknowledgments

We would like to thank François Armand and Frédéric Herrmann for their important influence on the design described here.

Hugo Coyotte, Corinne Delorme, Pierre Lebée and Pierre Léonard ported the PVM to several architectures. They provided useful feedback, particularly on the portability aspect. Ivan Boule, Jean-Jacques Germond, Sabine Habert, Sylvain Langlois, Marc Maathuis, Denis Metral-Charvet, Laurence Mosseri, François Saint-Lu, Eric Pouyoul, Eric Valette, were the first, patient and exacting users of the Chorus memory management system.

Michel Gien, Marc Guillemont, Claude Kaiser and Will Neuhauser supplied many ideas improving of this presentation. Hubert Zimmermann, leader of the Chorus-systèmes industrial venture, made all this possible.

Finally, we thank Richard Rashid for communicating benchmark results on Mach.

References

[1] V. Abrossimov, M. Rozier, and M. Gien. Virtual Memory Management in Chorus. In *Lecture Notes in Computer Sciences, Workshop on Progress in Distributed Systems Management*, Springer-Verlag, Berlin (Germany) April 1989.

[2] F. Armand, M. Gien, F. Herrmann and M. Rozier. Revolution 89 or "Distributing Unix Brings it Back to its Original Virtues". In *Proc. "Workshop on Experiences with Building Distributed (and Multiprocessor) Systems"*, Ft. Lauderdale FL (USA), October 1989.

[3] D.G. Bobrow et al. TENEX, a paged time sharing system for PDP-10. *Communications of the ACMxx*, 15(3), 1972.

[4] David R. Cheriton. The Unified Management of Memory in the V Distributed System. Technical Report, Computer Science, Stanford University CA (USA), 1988.

[5] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. In *Proc. USENIX Summer'87 Conference*, Phoenix AR (USA), June 1987.

[6] Frédéric Herrmann, François Armand, Marc Rozier, Michel Gien, Vadim Abrossimov, Ivan Boule, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser. Chorus, a new technology for building Unix systems. In *Proc. EUUG Autumn '88 Conference*, Cascais (Portugal), October 1988.

[7] Paul J. Leach, Paul H. Levine, James A. Hamilton, and Bernard L. Stumpf. The file system of an integrated local network. In *ACM Computer Science Conference*, New Orleans LA (USA), March 1985.

[8] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proc. Principles of Distributed Computing (PODC) Symposium*, pages 229–239, 1986.

[9] Jose Alves Marques, Roland Balter, Vinny Cahill, Paulo Guedes, Neville Harris, Chris Horn, Sacha Krakowiak, Andre Kramer, John Slattery, and Gerard Vendôme. Implementing the Comandos architecture. In *Esprit'88: Putting the Technology to Use*, pages 1140–1157, 1988 North-Holland.

[10] Régis Minot, Pierre Courcoureux, Hubert Zimmermann, Jean-Jacques Germond, Paolo Alvari, Vincenzo Ambriola, and Ted Dowling. The spirit of Aphrodite. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), pages 519–539, November 1988.

[11] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. In *ACM Transactions on Computer Systems*, 6(1), February 1988.

[12] Michael N. Nelson and John K. Ousterhout. Copy-on-write for Sprite. In *Proc. Summer Usenix '88 Conf.*, San Francisco CA (USA), pages 187–201, June 1988.

[13] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[14] Marc Rozier and José Legatheaux-Martins. The Chorus distributed operating system: some design issues. In *Distributed Operating Systems, Theory and Practice*, Springer-Verlag, Berlin, 1987.

[15] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Pierre Léonard, Sylvain Langlois, and Will Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4), 1988.

[16] Marc Shapiro. The design of a distributed object-oriented operating system for office applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[17] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proc. 6th IEEE Int. Conf. on Distributed Computing Systems*, Cambridge, MA (USA), May 1986.

[18] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black and Robert Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proc. 11th ACM Symp. on Operating Systems Principles*, Austin TX (USA), November 1987.

[19] Hubert Zimmermann, Jean-Serge Banino, Alain Caristan, Marc Guillemont, and Gérard Morisset. Basic Concepts for the Support of Distributed Systems: the CHORUS approach. In *Proc. 2nd IEEE Int. Conf. on Distributed Computing Systems*, Versailles (France), April 1981.