[14] 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.

[15] 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.

References

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic Virtual Memory Management for Operating System Kernels Technical Report CS/TR-89-18, Chorus systèmes, 1989. Sumbitted for publication.

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

[3] 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.

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

[5] 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.

[6] Régis Minot, Pierre Courcoureux, Hubert Zimmermann, Jean-Jacques Germond, Paolo Alvari, Vincenzo Ambriola, and Ted Dowling. The spirit of Aphrodite. In *Esprit'88: Putting the Technology to Use*, pages 519–539, 1988 North-Holland.

[7] 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.

[8] 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.

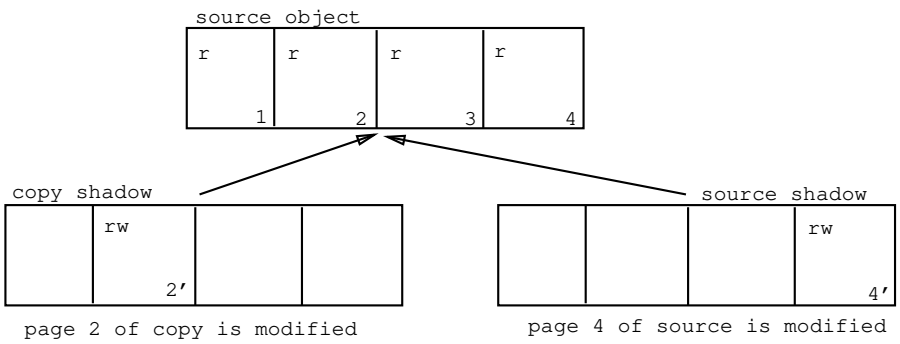
[9] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[10] 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.

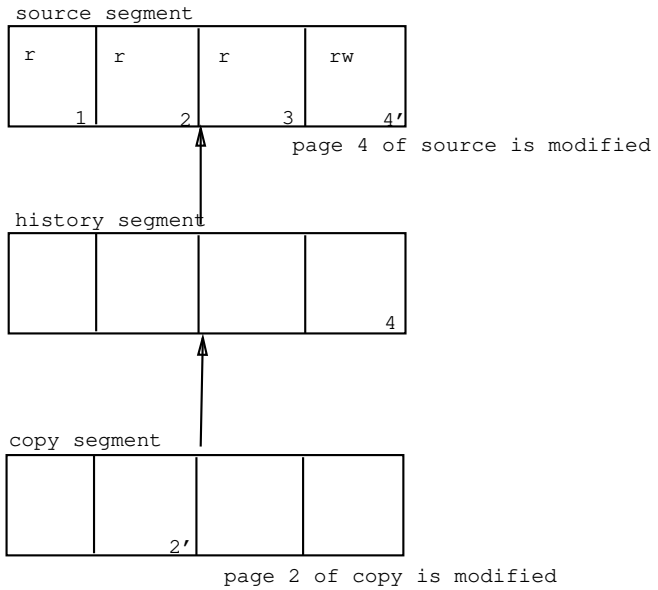
[11] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4), 1988.

[12] Marc Shapiro. The design of a distributed object-oriented operating system for office applications. In *Esprit'88: Putting the Technology to Use*, 1988 North-Holland. November 1988.

[13] 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.



shadow objects technique



history objects technique

Figure 4: Shadow and History objects

RPC operation, or when the receiver of the message is local and is already waiting for the message.

3.7 Deferred copy

In the implementation of the rgnInit, rgnInitFromActor, rgnCopy, sgCopy, sgRead and sgWrite operations the *history objects* deferred-copy technique is used.

Our technique was inspired by the Mach's shadow objects [9]. When Mach initializes a segment (which is called a *memory object*) as a copy of another, the source is set read-only, and two new memory objects, the shadow objects, are created (see Figure 4). The two shadows keep the modified pages of the source and the copy objects respectively; the original pages remain in the source object. So, the current state of the source and the copy are dispersed across two objects.

On the other hand, when Chorus initializes a segment as a copy of another, two new segments are created: the copy and the *history*. The copy keeps only its own modified pages, the history is to keep the original pages modified in the source segment which always keeps its current pages (modifed or not). So, the current state of the source is never dipersed, but the state of the copy may be dispersed across three segments. The advantages and performance of the history objects technique are described in detail in [1].

4 Conclusion

Like the other kernel services, the Chorus virtual memory management service has been designed as a set of basic tools, suited for versatile implementations of various system policies. In particular, memory objects are managed outside the kernel, by user-level servers. The tools provided by the kernel allow these servers to manage object caching and cache consistency with their own specific policies. All the interactions between these servers and the intra-kernel memory management services are performed via the Chorus network-transparent IPC: these servers may be distributed as needed, allowing a great variety of configurations.

The Chorus virtual memory management has been designed to be highly portable on a wide range of modern hardware architectures. It is mostly written in C++, and a small hardware-dependent part is clearly isolated from the main machine-independent part. The Chorus kernel has 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 SPARC, MC88000 and ARM-3 based machines.

5 Acknowledgments

We wish to thank Marc Shapiro of INRIA for his ideas on virtual memory management, and for helpful comments on early drafts of this paper. Many thanks also to Francois Armand, Hugo Coyote, Corinne Delorme, Marc Guillement, Fréderic Herrmann, Pierre Lebée and Pierre Léonard contributed, each with a particular skill, to the Chorus virtual memory specification and implementation on various machine architectures.

The chSync operation forces all modified portions of a local cache fragment to be written back (by a mpPushOut) to the segment. The chFlush operation releases a local cache fragment; modified portions of the fragment are first written back to the cached segment. The chInvalidate operation destroys a local cache fragment, without any synchronization.

The operation chLockInMemory permits a fragment to be fixed in real memory (so that it cannot be thrown away or flushed by the kernel).

The chRelease operation demands that the kernel destroy a local cache. The destruction will be refused if the local cache is currently mapped into a region.

3.5 Segment caching

The kernel recognizes two basic types of segments: *temporary* and *permanent*. The kernel itself requests the creation of temporaries (see section 3.4.3); these correspond to "swap" areas for program data. Permanent segments correspond to user objects.

When some segment is no longer in use, the corresponding local cache could be discarded. Instead, the kernel keeps such an unreferenced local cache (of a permanent segment) as long as possible, i.e. as long as there is enough free physical memory, and enough space in the kernel tables. When a program requests the use of a permanent segment, the kernel first checks to see if there is already a local cache 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.

3.6 IPC implementation

IPC messages serve to transport data, both for users and for the system. Therefore we have tried to 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 [2] than to Mach [14].

Messages are of limited size (64 Kbytes maximum 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.

When a message is sent, the kernel transfers its body through a temporary segment called *IPC buffer* using sgWrite operation. The kernel manages the IPC buffer as a pool of fixed-sized (64 Kbyte) slots. The number of slots is not limited: the IPC buffer is not mapped in the kernel space and is never locked entirely in physical memory. When a message is received by a thread, the kernel transfers the message body from the IPC buffer to the thread's context, using the sgRead operation. In order to avoid the data copy the move option of data transfer is used if possible.

There is one IPC buffer per site. When a message must be sent to another site, the network manager is in charge of transferring the message body between the IPC buffers. The sgLockInMemory operation can be used to fix a slot into real memory as it is being transmitted over the network.

Future optimisations will concentrate on avoiding the extra message transfer, from the source actor address space to the IPC buffer, when the message is sent by an synchronous

Mapper operations
mpUsed (segment, localCache) *segment will be used through a cache*
mpPullIn (segment, localCache, offset, size, accessMode) → (data, realSize) *read a fragment of the segment*
mpGetWriteAccess (segment, localCache, offset, size) *request write access to a fragment*
mpPushOut (segment, localCache, offset, data, size) *write a fragment back*
mpCreate (mapper, localCache) → segment *create a segment that will be used through a cache*
mpRelease (segment, localCache) *end access to a given segment via the cache*

Table 3: Mapper Interface.

Local cache control
chSync (localCache, offset, size) *synchronize a local cache fragment*
chInvalidate (localCache, offset, size) *destroy a local cache fragment*
chFlush (localCache, offset, size) *synchronize and invalidate a fragment*
chLockInMemory (localCache, offset, size) *fix a local cache fragment in real memory*
chUnlock (localCache, offset, size) *permit a local cache fragment to be flushed*
chRelease (localCache) → ok *release the local cache, if possible*

Table 4: Local Cache Control Interface.

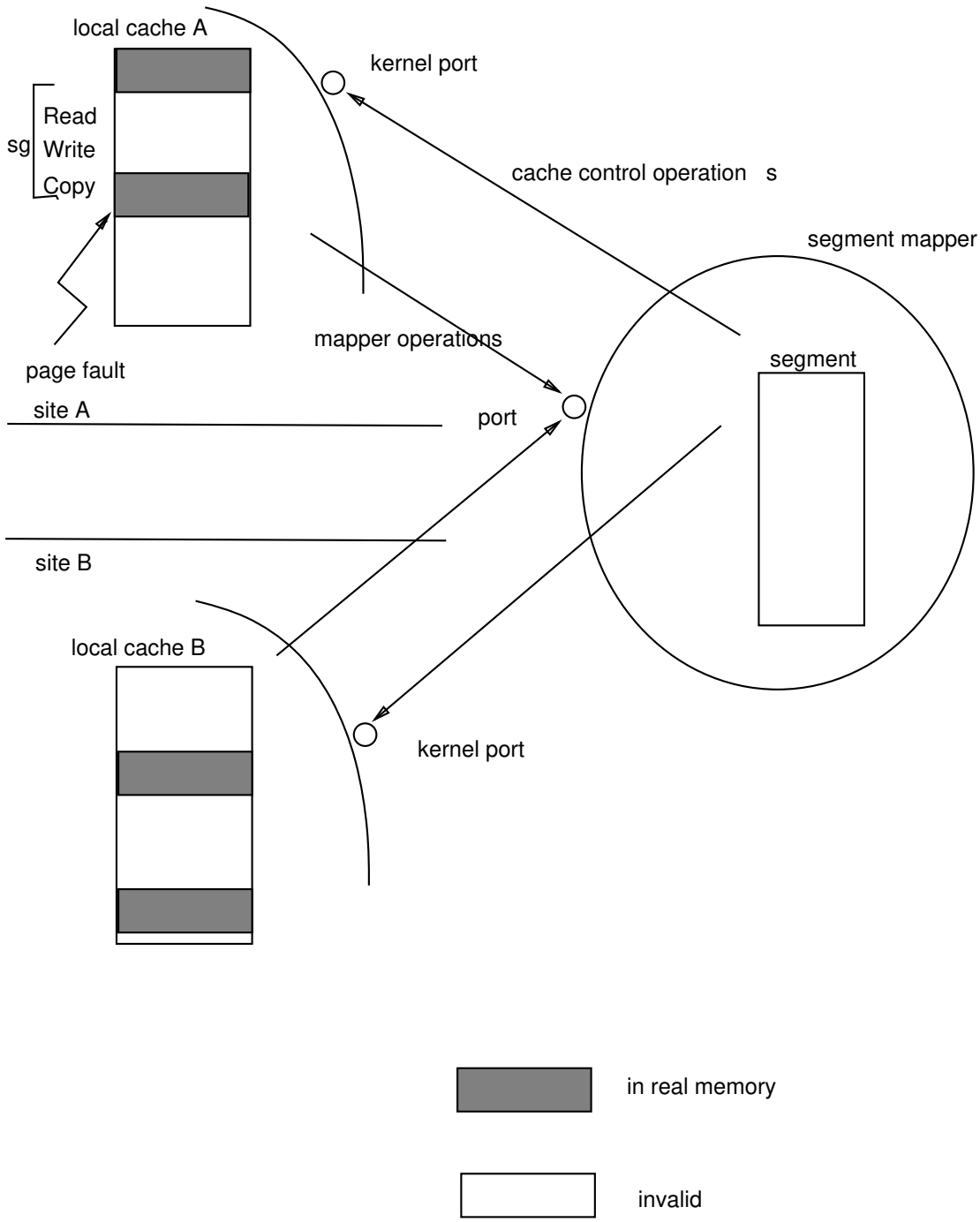


Figure 3: Local Caches

3.4 External mappers

3.4.1 Segment capability

Segments are designated by **capabilities**, containing the mapper's port name and a key. The key is opaque for the system, and used by the mapper to manage and protect segment access.

For example, a Unix file server can construct a capability for a Unix file, by concatenating the file server port name, the file inode number and a cryptographic protection key.

3.4.2 Local caches

When the kernel decides (e.g. on a page fault or a segment operation) to make available a fragment of a segment in the form of physical memory, it extracts the segment mapper port name from the segment capability, and sends the mpPullIn request to the mapper, using the Chorus RPC mechanism. The mapper responds with a message containing the data.

The kernel encapsulates the physical memory holding portions of the segment data in a per segment **local cache** object (see Figure 3).

The same cache object is used for both the mapped and the explicit segment access, thus resolving the double-caching problem (see section 3.3).

A local cache object is designated by its capability; the server for local caches is the kernel (see section 3.4.4). Using the local cache capability, a mapper is able to distinguish between different local caches, on the different sites, of the same segment, and to implement distributed consistency maintainance protocols.

3.4.3 Mapper operations

Table 3 describes the operations, that a local cache object (in fact the kernel) may invoke on the corresponding segment mapper. The mapper is always invoked using the Chorus RPC mechanism. Any user–level segment mapper server must export all these functionals, (except for the mpCreate operation, which is exported only by the default mapper).

The mpUsed operation is invoked by the kernel when a new local cache is created. When the kernel destroys a local cache, it signals this action to the mapper with mpRelease.

When the kernel needs to fill up a fragment of the local cache, the mpPullIn operation is performed. Cached data carries the access rights defined by the accessMode argument; when a write access occurs to data which is cached read-only, the kernel invokes the mpGetWriteAccess to request write access. When the kernel needs to save a fragment of cached data, it calls the mpPushOut operation on the corresponding segment.

When the kernel needs to create a new temporary segment (e.g. on rgnAllocate or rgnInit operations) it performs the mpCreate operation on the default mapper.

3.4.4 Local cache control

Table 4 describes the operations that a mapper may perform to a local cache. This interface is sufficient for a mapper to implement a distributed virtual memory consistency maintainance protocol [4].

Segment operations
sgCopy (segment, offset, srcSegment, srcOffset, size, move) *copy data from one segment to another segment*
sgRead (segment, offset, dstActor, dstAddress, size, move) *copy data from a segment to a region*
sgWrite (segment, offset, srcActor, srcAddress, size, move) *copy data from a region to a segment*
sgLockInMemory (segment, offset, size) *lock a fragment of a segment into physical memory*
sgUnlock (segment, offset, size) *permits a fragment of a segment to be swapped out*

Table 2: Segment Access Interface.

previously in real memory. For instance, the rgnLockInMemory and rgnUnlock operations can be used by device drivers to fix buffers in the physical memory during an I/O.

The rgnCopy operation permits data transfer between two existing regions. When the move flag is set, the contents of the source fragment are undefined after the data transfert. This permits the data to be remapped, instead of performing a copy or copy-on-write. This operation can be used, for instance, by the Unix sub-system to avoid extra copies from the source process address space to an intermediate buffer during read/write operations on pipes or sockets.

The rgnFree operation deallocates the region containing the specified virtual address.

3.3 Segment operations

The table 2 describes explicit-access interface for segments.

The sgCopy, sgRead and sgWrite operations copy a fragment of one segment to another segment. A segment can be described by its capability or by a virtual address in the actor in which it has been mapped. When the move flag is present, the contents of the source fragment are undefined after the data transfer, thus allowing the data to be remapped instead of copying it. These operations could be used by the Unix sub-system to implement read/write operations on files.

In a Unix-like system with demand-paging, there are two potential conflicts between read/write and mapped access to segments. First, the file buffers and the page buffers conflict for the real memory allocation, which can lead to a poor use of real memory. Secondly, the double-caching problem: if a segment can be both mapped and read/written, and each access has its own cache, the two caches can become inconsistent. In Chorus a given segment may be mapped in a region and, at the same time, be accessed by explicit operations. The underlying representation (see section 3.4.2) avoids the above-mentioned conflicts.

The semantics of the sgLockInMemory and sgUnlock operations are the same as rgnLockInMemory and rgnUnlock respectively.

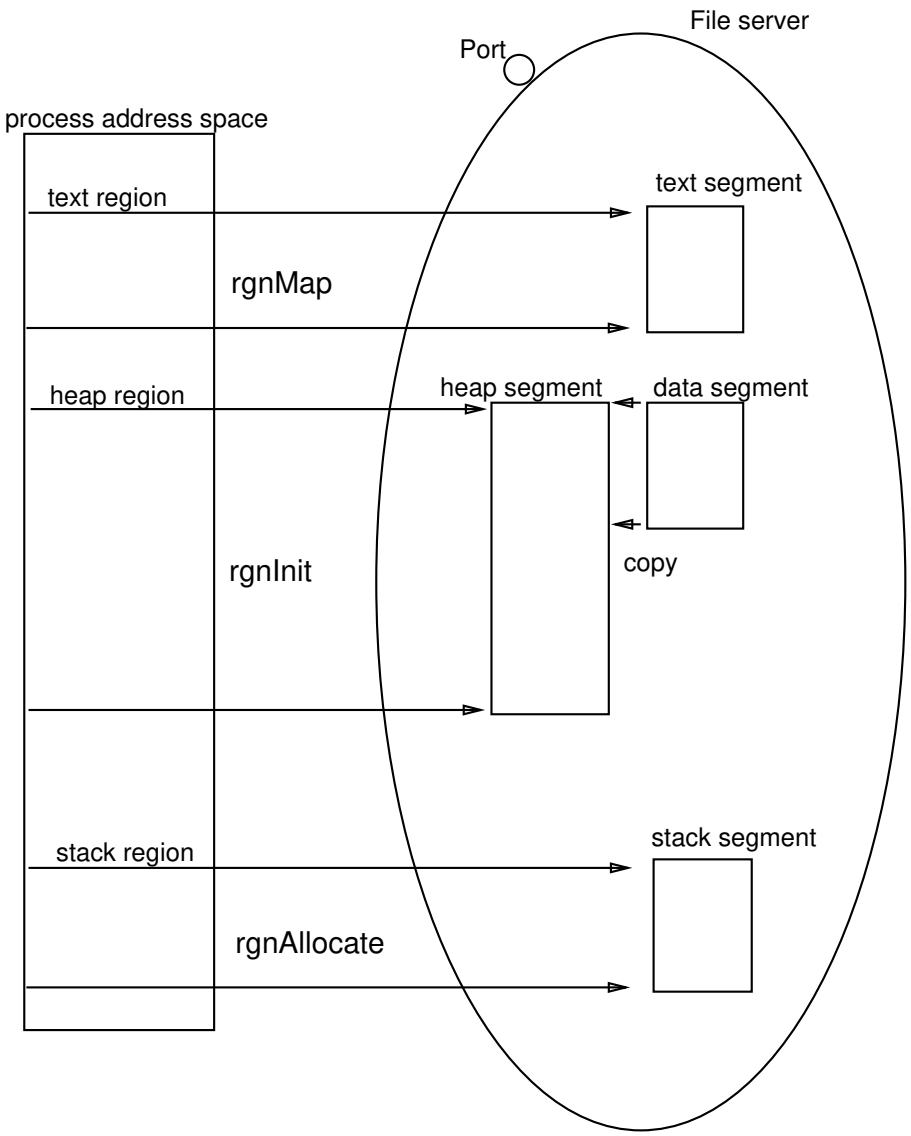


Figure 2: Exec of an Unix process

Context management
rgnAllocate (actor, address, size, protections) *map a region to a scratch temporary segment*
rgnMap (actor, address, size, protections, segment, offset) *map a region to a segment*
rgnInit (actor, address, size, protections, srcSegment, srcOffset, srcSize) *map a region to a temporary segment, initialized from another segment*
rgnMapFromActor (actor, address, size, protections, srcActor, srcAddress) *map a region to the segment mapped to another region*
rgnInitFromActor (actor, address, size, protections, srcActor, srcAddress, srcSize) *map a region to a temporary segment* *initialized from the segment mapped to another region*
rgnCopy (actor, address, srcActor, srcAddress, size, move) *copy data from one region to another*
rgnLockInMemory (actor, address, size) *lock a fragment of a region into physical memory*
rgnUnlock (actor, address, size) *permits a fragment of region to be swapped out*
rgnFree (actor, address) *deallocate a region*

Table 1: Context Management Interface.

The rgnAllocate operation creates a new temporary segment and maps it into the context. The initial content of the segment is not defined. The real segment creation is delayed to the first mpPushOut (see section 3.4.3) operation on the segment. The rgnMap operation maps an existing segment to a region. The rgnInit operation creates a new temporary segment, initializes it from another segment and maps it into the context. The source segment is described by its capability. For instance, when the Unix sub-system creates a process from a file during an exec (see Figure 2), three regions are created: the text region, using rgnMap; the process initialized data, bss and heap, using rgnInit; and the process stack, using rgnAllocate. The file system exports two capabilities per executable file: one describing the process text and the other the process initial data.

The rgnMapFromActor and rgnInitFromActor operations have the same semantics as rgnMap and rgnInit respectively, on except that the segment is not described by its capability but rather by its address within an actor in which it has been already mapped. For instance, when the Unix sub-system creates a process during a fork, three regions are created in the child process for the text, data and stack. The text region is created using rgnMapFromActor to share the parent's text segment; the data and stack regions are created using rgnInitFromActor, to intialize the child's data and stack segments from the parent's.

The rgnLockInMemory operation permits a region fragment [5] to be fixed in real memory, so that it cannot be swapped out by a mpPushOut (see section 3.4.3) operation. The rgnLockInMemory causes mpPullIn operations to occur, for any portions of the fragment not

[5] Here and subsequently, we use the term *fragment* to name a portion of a region (segment) specified by its starting address (offset) within the region (segment) and its size.

users must know in order to be able to modify the object. When an object is managed by some external server (e.g. segments, see section 3.1), the UI is the global name of a port of that server, and the key identifies the object within the server and holds the protection information. The semantics of keys are defined by the servers. As UI's, capabilities may be freely exchanged in messages: the kernel does not control their transmission.

3 Chorus Virtual Memory

The Chorus memory management service provides:

- separate address spaces (if the hardware gives adequate support), associated to actors. We will use the term **context** to name an address space in the remainder of this paper.
- efficient and versatile mechanisms for data transfer between contexts, and between secondary storage and a context. The mechanisms are adapted to various needs, such as IPC, file read/write or mapping, memory sharing between contexts, and context duplication.

3.1 Basic abstractions

Chorus memory management considers the data of a **context** to be a set of non-overlapping **regions**, which form the valid portions of the context. These regions are mapped (generally) to secondary storage objects, called **segments**.

Segments are managed outside of the kernel, by external servers called segment **mappers**. These manage the implementation of the segments, as well as the protection and naming of segments. They export a simple segment access interface (described in section 3.4.3) to the kernel. The subsystem running on top of the kernel must provide at least one *default mapper* to permit the kernel to create temporary segments (e.g. "swap" segments).

A region may map a whole segment, or part of one, in which case it serves as a window into the segment; the window may be caused to slide for sequential access. Protection flags (e.g. read/write/execute, user/system) are associated with each entire region. Access to different parts of a segment can be protected differently, by mapping each to a separate region.

In addition to the mapped-memory access described above, the same segment can be accessed by explicit data transfer, as decribed in section 3.3.

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 explicit operations, by any number of threads.

The consistency of a segment shared among actors of the same site is guaranteed by the kernel, but when a segment is shared among different sites, the segment mapper is in charge of maintaining the segment consistency, using mechanisms described in section 3.4.4.

3.2 Context management

The table 1 describes context management operations.

that actor and no other actor. Threads are scheduled by the kernel as independent entities. In particular, threads of an actor may run in parallel on the many processors of a *multiprocessor* site.

The threads making up an actor can communicate and synchronize using the shared memory provided by the actor address space. In addition, Chorus offers message-based utilities which allow any thread to communicate and synchronize with any other thread, on any site. This is known as *IPC (Inter-Process Communication)*. The Chorus IPC permits threads to exchange messages either *asynchronously* or by *demand/response*, also called *Remote Procedure Call (RPC)*.

The principle characteristic of the Chorus IPC is its transparency with respect to the location of threads: the communication interface is uniform, regardless of whether it is between threads in a single actor, between threads in different actors on the same site, or between threads in different actors on different sites.

A **message** is an untyped string of bytes, of variable but limited size[3], called the *message body*. The *sender* of the message may optionally join a fixed size [4] string to the message body, the *message annex*. When present, the *annex* is copied from the *sender* address space to the *receiver* address space. By default, the *message body* is transferred with *copy semantics*. However, options are available to allow the transfer of the body without copying (see 3.6).

Messages are not addressed directly to threads, but to intermediate entities called **ports**. The port is an address to which messages can be sent, and a queue holding the messages received but not yet consumed by the threads. For a thread to be able to consume the messages received by a port, it is necessary that this port be *attached* to the actor that supports the thread. A port can only be attached to a single actor at a time, but can be successively attached to different actors, effectively *migrating* the port from one actor to another. This migration can be accompanied, or not, by the messages queued on the port.

The notion of a port provides the basis for dynamic reconfiguration: this extra level of indirection between communicating threads, enables a given *service* to be supplied independently of a given actor. The servicing actor can be changed at any time, by changing the attachment of the port from the first thread's actor to the new thread's actor.

A **group** of ports connects those ports to a *multicast* facility: either from one thread to an *entire group of threads* (via a group of ports); or "functional" access to a service: a server is selected from a group of (equivalent) servers. A group of ports is essentially a name to which messages can be addressed. A group is built by dynamically inserting ports into, and removing them from, the group.

Ports are globally designated with **Unique Identifiers** (UI's). A UI is unique in a Chorus network. The Chorus kernel implements a location service, allowing threads to use these names without knowledge of the locality of the designated entities. UI's may be freely exchanged in messages; the kernel does not control their transmission.

The global names for other types of objects are based on UI's, but hold more information, such as protection information. These names are called **capabilities** [13]. A capability is made of a UI and an additionnal structure, the *key*. When objects are kernel objects (e.g. actors), the UI is the global name for the object, and the *key* is only a protection key, that

[3] 64 Kbytes
[4] 64 bytes

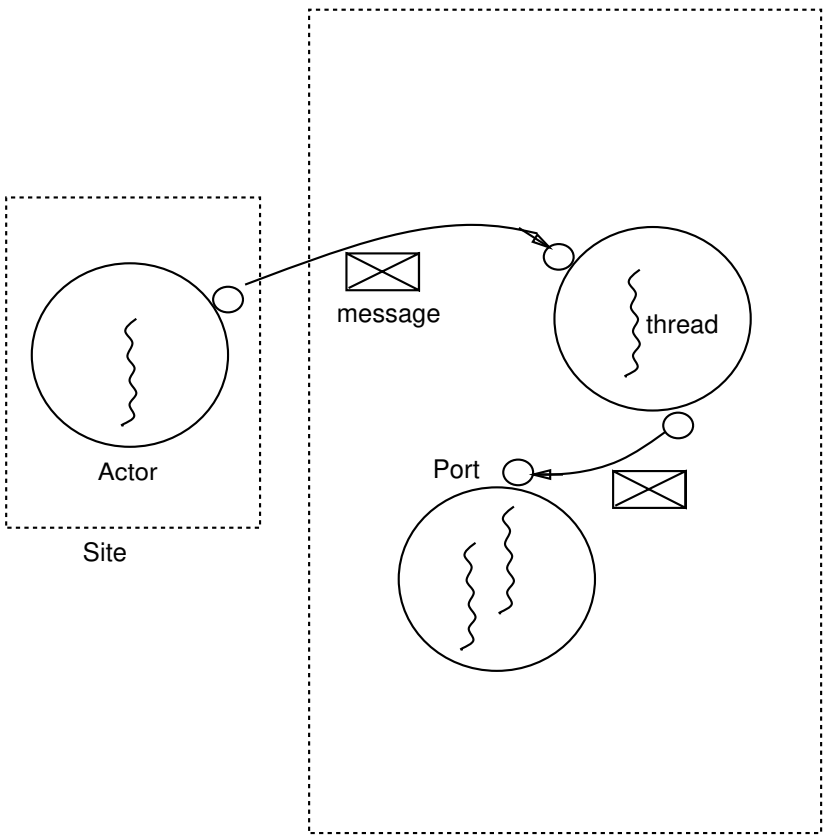


Figure 1: Actors, threads, ports and messages

1 Introduction

The Chorus[1] technology has been designed for building new generations of open, distributed, scalable operating systems.

Chorus is a communication-based technology. Its minimal kernel integrates distributed processing and communication at the lowest level. Chorus operating systems [11] are built as sets of independent system servers, to which the kernel provides the basic services such as activity scheduling, network transparent IPC, memory management and real-time event handling. The Chorus kernel can be scaled to exploit a wide range of hardware configurations, such as small embedded boards, workstations or high-performance servers. Operating systems (called *subsystems*) implemented on top of this kernel currently include a full Unix System V[2] [3] and PCTE [6]. Work is currently in progress to implement object-oriented distributed subsystems [12, 5].

CHORUS-V3 is the current version of the Chorus Distributed Operating System, developed by Chorus systemes. Earlier versions were studied and implemented within the Chorus research project at INRIA between 1979 and 1986.

This paper focuses on the description of the virtual memory management service provided by the Chorus kernel. Due to the multiple purposes of the Chorus kernel, its memory management service has been designed as a well-isolated component, offering generic interfaces adapted to various hardware architectures and to various system needs [1]. In particular, the secondary storage objects are managed outside the kernel, within independent system servers.

The outline of the rest of this paper is the following. In section 2 we briefly present an overview of the basic Chorus kernel abstractions. In section 3, we describe the virtual memory management service: its abstractions, interfaces, implementation issues, and some examples of the use of the virtual memory management interface by our Unix implementation.

2 Chorus basic abstractions

The physical support for a Chorus system is composed of a set of **sites** ("machines" or "boards"), interconnected by a communications *network* (in a general sense: either network or bus). A site is a tightly coupled grouping of physical resources: one or more processors, central memory, and attached I/O devices. There is one Chorus kernel per site.

The **actor** (see Figure 1) is the unit of distribution in the Chorus system. An actor defines a protected address space supporting the execution of **threads** which share the address space of the actor. Any given actor is tied to a site, and its threads are executed on that site. A given site can support many simultaneous actors.

The **thread** is the "unit of execution" in a Chorus system. A thread is a sequential flow of control and is characterized by a *context* corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.). A thread is always tied to one and only one actor. The actor constitutes the execution environment of the thread. Within the actor, many threads can be created and can run in parallel. These threads share the resources of

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

List of Figures

1 Actors, threads, ports and messages 2
2 Exec of an Unix process 6
3 Local Caches 9
4 Shadow and History objects 13

List of Tables

1 Context Management Interface. 5
2 Segment Access Interface. 7
3 Mapper Interface. 10
4 Local Cache Control Interface. 10

Contents

List of Figures ii

List of Tables ii

1 Introduction 1

2 Chorus basic abstractions 1

3 Chorus Virtual Memory 4

3.1 Basic abstractions 4

3.2 Context management 4

3.3 Segment operations 7

3.4 External mappers 8

3.4.1 Segment capability 8

3.4.2 Local caches 8

3.4.3 Mapper operations 8

3.4.4 Local cache control 8

3.5 Segment caching 11

3.6 IPC implementation 11

3.7 Deferred copy 12

4 Conclusion 12

5 Acknowledgments 12

© Chorus systèmes, 1989

Virtual Memory Management in Chorus

Vadim Abrossimov, Marc Rozier and Michel Gien

Chorus systèmes, 6, ave. Gustave Eiffel, 78182 Saint-Quentin-en-Yvelines cedex (France)
Tel: +33 1 30 57 00 22, Fax: +33 1 30 57 00 66, E-mail: mr@chorus.fr

Abstract

The Chorus technology has been designed for building "new generations" of open, distributed, scalable operating systems. It is based on a small kernel onto which operating systems are built as sets of distributed cooperating servers. This paper presents the Virtual Memory Management service provided by the Chorus kernel. Its abstractions, interfaces and some implementation issues are discussed. Some examples of the use of this interface by our distributed Unix implementation are given.

In: Proceedings of the
"Progress in Distributed Operating Systems and Distributed Systems Management"
workshop, Berlin, 18-19 April 1989
Lecture Notes in Computer Science, Springer Verlag.

© Chorus systèmes, 1989