Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques

Article *in* ACM SIGPLAN Notices · October 1989

DOI: 10.1145/74878.74905

Roy Campbell
University of Illinois, Urbana-Champaign

Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques*

Vincent F. Russo and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave., Urbana, IL 61801-2987

Abstract

The *Choices* operating system architecture [?, ?, ?] uses class hierarchies and object-oriented programming to facilitate the construction of customized operating systems for shared memory and networked multiprocessors. The software is being used in the Tapestry Parallel Computing Laboratory at the University of Illinois to study the performance of algorithms, mechanisms, and policies for parallel systems. This paper describes the architectural design and class hierarchy of the *Choices* memory and secondary storage management system.

The mechanisms and policies of a virtual memory system implement a memory hierarchy that exploits the trade-offs between response times and storage capacities. In *Choices*, the notion of a memory hierarchy is represented by layers in which abstract classes define interfaces between and internal to the layers. Concrete subclasses implement new algorithms or data structures or specializations of existing ones. This paper describes the motivation for an object-oriented, class-hierarchical approach to virtual memory system design, and describes the overall architecture of such an approach, as it has been applied to the *Choices* system. Special attention is paid to the advantages in both design and implementation that have resulted from using object-oriented techniques.

*This work was supported in part by NSF grant number CISE-1-5-30035, by NASA NSG1471 and by AT&T METRONET.

1 Introduction

The *Choices* operating system architecture [?, ?, ?] is motivated by the difficulties of building portable and extensible operating systems for high-performance multiprocessor and uniprocessor computers. The conventional operating system provides applications with a "kernel" offering a predefined selection of system services that cannot be easily extended to provide specialized or user defined services. *Choices* uses object-oriented programming and class hierarchies to organize and facilitate solutions to this problem as a *family* of systems. A member of this family can be built by specializing classes in the hierarchy. This leads to the notion of a "customized" operating system. *Choices* is designed to be an object-oriented system from the "ground up", i.e. from the hardware through the application level.[1] Objects and classes encapsulate all internal data structures and algorithms of the operating system. Both internal services for system management and external services for application support are provided by object method invocation.

This paper presents the *Choices* virtual memory and backing storage management system and discusses how object-oriented design and programming techniques were used in its implementation. Aside from the presentation of the system itself, examples are given of ways the object-oriented paradigm has aided both the design and implementation of the system. This paper is, therefore, not simply a description of a particular object-oriented system or a discussion of its implementation. Rather, it is an attempt to show by example the benefits to operating system design and construction that can be gained

[1] We feel this is important. The low level details necessary to implement such a system provide another test of the usefulness of object-oriented programming and have allowed *Choices* to provide a counter example to the "myth" that object-oriented systems are less efficient that traditional systems.

via object-oriented techniques.

Before detailing the *Choices* memory management classes, a brief overview of the general *Choices* class hierarchy is appropriate. Instances of class *Process* are the basic units of execution in a *Choices* system. A process is represented by the information it needs to execute. This includes a copy of its processor state (an instance of the *MachineContext* class), and a description of its virtual memory (a reference to an instance of the *Domain* class). Processes are scheduled and executed within a *Choices* system by being transferred between *ProcessContainers* via the *add* and *remove* methods. Subclasses of ProcessContainer implement particular scheduling disciplines via these methods. The *CPU* subclass of ProcessContainer supports process execution. Multiple instances of the CPU class are used to represent multiprocessor architectures. The *Exception* class and its subclasses provide for exception handling, including traps and interrupts. The raising of an exception, via invoking the *raiseException* method on a CPU object (with the Exception to raise as an argument), usually results in the movement of Processes between ProcessContainers.[2] The *MemoryObject* class provides support for logical, or secondary storage management in *Choices*. Finally, the *AddressTranslation* class encapsulates hardware dependent virtual-to-physical address conversion mechanisms.

This paper concentrates on the classes within *Choices* which support virtual memory and backing storage management. The *Choices* design exploits virtual memory techniques for efficient interprocess communication via shared memory. Any communication required between applications is supported directly by operations on shared objects or directly via shared virtual memory. *Choices* support for networked multiprocessors extends the virtual memory across the network[?].[3]

In particular the *Choices* design seeks to support:

- Efficient sharing of memory between processes executing in parallel.
- Efficient context switching between interrupt and user processes.
- Virtual memory spaces that are larger than physical memory.
- Multiple, arbitrary, backing stores.
- Access to objects whose lifetimes exceed the lifetime of an individual process or its virtual memory.
- Efficient process creation and message passing primitives that only copy shared memory objects (such as data or code) when necessary.[4]
- Appropriate page replacement algorithms for memory objects.[5]
- Uniform and consistent memory management and buffering schemes that can be applied to virtual memory, such as input/output buffering, files and file-buffer caches, and message caches.

After discussing background and related research, the virtual memory and backing storage management approach adopted in *Choices* is introduced. Next, a class hierarchy that implements this approach is outlined along with the major methods of each of the classes in the hierarchy.

2 Background

Most modern computer architectures provide hardware support for virtual memory. Operating systems use virtual memory to provide support for large address spaces on systems with limited physical memory, to provide protection both within and between applications, to provide data sharing, and to provide artificial contiguity. Virtual memory systems include three major components: an address translation mechanism, a virtual memory placement algorithm, and a virtual memory *re*placement algorithm.

In a system supporting virtual memory, each executing process has an associated translation table that maps valid virtual memory addresses to physical memory addresses. *Dynamic address translation* hardware performs this mapping function. To reduce the number of memory references, most hardware maintains an associative cache of memory mappings. Virtual memory systems simplify physical memory allocation by partitioning the address space of a process into fixed-size pages, or variable-size segments [?, ?].[6]

[2]The process and exception management systems of *Choices* are discussed in detail in [?].

[3]Message-oriented kernels like the V Kernel [?], Accent [?], Amoeba [?], and MICROS [?] build specific communication schemes into the lowest levels of the kernel. For example, some systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easily adaptable to support other approaches such as "send process on read" or "remote procedure call on execute."

[4]That is, they employ copy-on-write shared memory techniques to minimize unnecessary copying.

[5]Most page replacement schemes in virtual memory management systems are global. Instead, *Choices*, allows localized page replacement schemes where each memory object may have its own algorithm that optimizes the paging traffic for that type of memory object.

[6]Two-level page table or segmented paging schemes may be employed in which contiguous pages are grouped into larger

The contents of a virtual memory are saved on secondary storage such as a disk. This is usually termed the *backing store* for the memory. The contents are accessed by transferring portions of them into physical memory. At first, the translation table address entries for portions of the virtual memory located on backing store are all marked "non-resident". A process attempting to access non-resident memory generates a fault. At this point, the virtual memory placement algorithm retrieves the contents from the backing store, places it in physical memory, and updates the address translation table. The instruction that caused the fault is then restarted (or continued).

Since the memory placement algorithm may eventually fill physical memory, the memory replacement algorithm exists to free physical memory. Non-modified data can just be discarded since they can be recovered from backing store. Modified data are returned to their backing store. As physical memory is freed, the corresponding translation table entries are marked non-resident. The memory replacement algorithm seeks to replace pages or segments so as to minimize I/O traffic between the main memory and the backing store. The optimal choice of data to replace is that which will not be referenced until the furthest time in the future[?].

3 The Memory Management Layers

The virtual memory and backing storage management system for *Choices* creates both an *object-oriented model* for the components of a virtual memory system and a *class hierarchy* that organizes variants of this model for different machines, architectures and applications.

Choices has adopted and extented some of the design ideas employed in the Mach [?] virtual memory management system. In particular, *Choices* adopts the idea of a *memory object* (an instance of the *Choices MemoryObject* class) that is cached in physical memory. *Choices* utilizes an object-oriented address translation layer which encapsulates the hardware. Existing algorithms[?] were refined and extended to implement the *Choices* object-oriented approach to the management of memory and secondary storage. *Choices* departs from other systems in its object-oriented, class hierarchical approach, allowing greater flexibility and customizability for given environments and applications.

The *Choices* storage and memory management system is divided into five conceptual layers (see Figure ??).

1. The address translation layer encapsulates the dynamic address translation hardware.

2. The physical memory layer allocates, frees and aggregates physical memory.

3. The logical memory layer provides access to a logically contiguous block of data stored on secondary storage.

4. The cached logical layer provides efficient access to logical memory by mapping portions of the logical memory to temporary physical memory locations.

5. The virtual memory layer maps the virtual address space of a process onto multiple logical memory backing stores, relying on the services of the cached logical layer to map them into physical memory.

3.1 Object-Oriented Programming

Object-oriented methodology encourages operating system *code reuse* within each layer, between layers, and across members of a family of related systems. The memory management layers of an operating system offer many opportunities for code reuse. Common data structures like queues, lists, hash tables, and system functions like memory allocation and deallocation routines often appear repeatedly within different layers.

Classes allow the reuse of the definition and implementation of abstract algorithms and data structures throughout the layers of a system. Subclassing allows refinement of these algorithms and data structures. This form of code reuse is enhanced by defining abstract classes to represent the abstract algorithms and data structures of useful operating system concepts, subclassing these classes for the specific requirements of an operating system, and then instantiating these classes to produce objects within a specific layer.

Many possibilities of code reuse exist within a family of operating systems. An abstraction of an operating system component used within many members of the family may be defined as an abstract class. Many subclasses each implementing a "version" of the

sections. A two-level page table is one in which a portion of the virtual address indexes into a first-level table containing pointers to second level tables. Another portion of the virtual address indexes a second level table and produces a pointer to the actual physical frame of memory containing the virtual page. The remainder of the virtual address is an offset from the start of the physical frame.

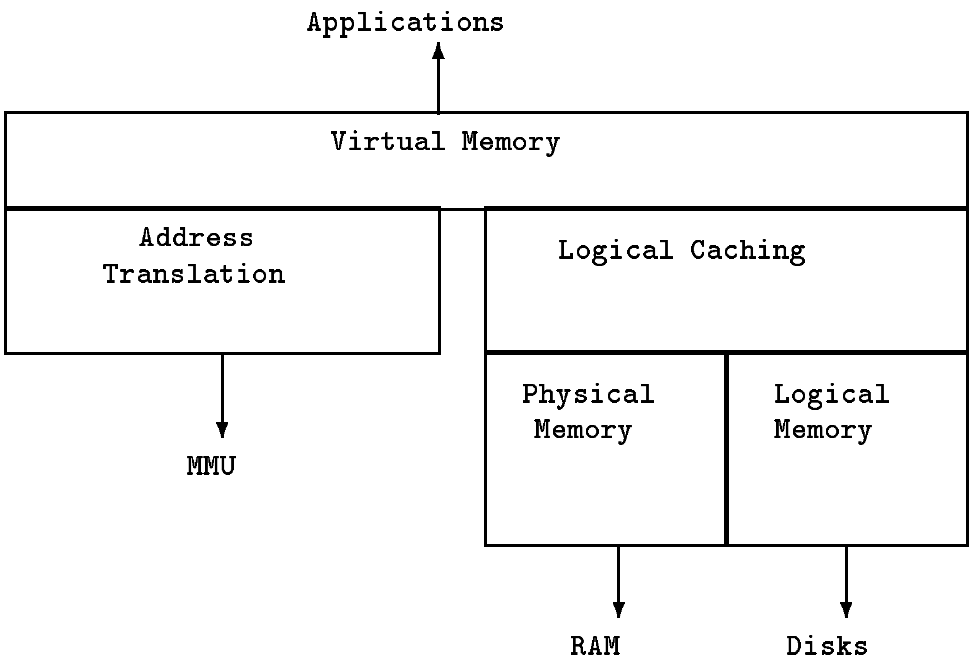

Figure 1: The *Choices* Memory Management Layers

class for particular hardware or application requirements may be created. If similar, the versions of the class can share a large portion of the implementation through inheritance.

The class hierarchical approach also encourages *customization*. Customization simplifies the generation of new members in the family of operating systems for new target machines and applications. Customization and modification of a family of systems is guided and aided by subclassing and by the structure imposed by the class hierarchy. The class hierarchy provides the systems designer with a conceptual view of how the operating system components function. It classifies these components with respect to their function. In addition, subclassing permits the behavior of a specific part of a *Choices* operating system to be modified without changing the rest of the system. In a well designed hierarchy, understanding the function of a parent class should permit the approximate function of a subclass to be inferred [?]. In the class hierarchy for *Choices*, only the top few classes need to be mastered to achieve a good overview of the interrelations between components of the system.

The following sections explain in detail each layer of the *Choices* virtual memory and backing storage management system and show how each of the above techniques have been applied to its design and construction.

4 The Address Translation Layer

Dynamic address translation hardware is the key mechanism that makes virtual memory practical. Across the spectrum of today's hardware, many different dynamic address translation mechanisms are employed. One of the major design goals of the *Choices* implementation is to ensure that the system remain portable and efficient despite this variety.

Dynamic address translation hardware schemes include multi-level page tables, inverted page tables (where a single entry represents each *physical* page and a hash of the virtual addresses is used to index into the table) and variable size pages. The machine dependencies can be encapsulated at the level of the page table entry that maps a single virtual page to a physical page or at that of the page table itself. However, not all dynamic address translation schemes use page tables; some use translation lookaside buffers [?].

The address translation layer provides the abstract mapping function between virtual and physical addresses in *Choices*. It is this function that is the purpose of the dynamic address translation hardware: to map a large, sparse virtual address space into physical memory addresses.

In *Choices* the address translation layer interface is defined by the abstract *AddressTranslation* class.

This class presents a *machine-independent interface* to the rest of the memory management system. Its function is to encapsulate the hardware-dependent representation of dynamic address translation. Its methods include: *addMapping*, to add a virtual-to-physical translation at a given protection level; *removeMapping*, to invalidate the mapping for a virtual address range; and *changeProtection*, to change the protection of a given range of virtual addresses.

When a request for a non-resident memory address occurs, machine-independent algorithms use machine-independent information to retrieve the data, update the appropriate AddressTranslation instance using addMapping, and continue. Machine-independent page or segment replacement algorithms swap out virtual memory pages or segments from main memory. The algorithms invoke the removeMapping method which invalidates the mapping.

The AddressTranslation class's changeProtection method exists to support the implementation of special-purpose virtual memory techniques such as *copy-on-write.* Such techniques require the ability to change the access allowed to a region of memory from *read-write* to *read-only* and back.

Following the example of the *pmap* system in Mach [?], machine-dependent representations of the virtual-to-physical memory mappings may be discarded at any time. They can be reconstructed from machine-independent information on demand. An AddressTranslation instance acts as a cache of currently active virtual-to-physical mappings. This information is stored in a *limited* amount of main memory dedicated to the machine-dependent dynamic address translation mechanism. An instance may discard mappings at any time in order to reuse portions of the memory dedicated to dynamic address translation. These mappings can be reconstructed from the machine-independent data on demand.

Concrete subclasses of AddressTranslation implement its methods for specific architectures. The AddressTranslation class hierarchy in *Choices* exemplifies code reuse across families of operating systems. *Choices* was first implemented on the National Semiconductor NS32332 [?] series of processors. This architecture uses four-kilobyte pages and a two-level page table to implement dynamic address translation. *Choices* represents this by the *NS32332Translation* class. *Choices* has also been customized to run on the Intel 80386 [?] processor. The 80386 and NS32332 architectures both use four kilobyte pages and a two-level page table. At the level of dynamic address translation, they differ only in the placement of a few bits in their respective page table entries.

The initial port to the 80386 was implemented by a person who was not involved with the initial design of *Choices*. The port included an *i386Translation* class. However, after a close comparison, the two implementations were combined to allow substantial code sharing.[7] A new *TwoLevelPageTable* class was introduced to define a generic two-level page table parameterized by the page size. The implementation of this class uses the auxiliary *PageTableEntry* class to represent an individual entry. Subclasses of PageTableEntry specify the representation of page table entries for the NS32332 (*NS32332PTE*) and the 80386 (*i386PTE*). The superclass TwoLevelPageTable collects common features of the original NS32332Translation and i386Translation implementations. The new NS32332Translation and i386Translation classes are subclasses of TwoLevelPageTable and reuse most of the original code via inheritance. Future support for other architectures with two level page tables should also be simplified by inheriting from the TwoLevelPageTable class. All that is needed is a new subclass of PageTableEntry. The page size parameter of the TwoLevelPageTable class trivializes a port to a system with a page size other than four kilobytes.

5 The Physical Memory Layer

Physical memory is a scarce resource in a virtual memory system and must be shared between the various applications. The *Choices* physical memory management layer allocates and deallocates physical memory to support the virtual memory system. The layer includes objects that maintain the status of physical memory blocks, manage the allocation and deallocation of physical memory, and aggregate blocks of physical memory into lists for simple manipulation.

5.1 PhysicallyAddressableUnit

An instance of the *PhysicallyAddressableUnit* class represents a block of *contiguous* physical memory. In paging systems, a block corresponds to a page. PhysicallyAddressableUnit instances are used to localize usage information about the block of memory represented. The *address* method returns the address of the physical memory. The *size* method returns the length of the block. The *referenced*, *setReferenced*, *modified*, and *setModified* methods access and

[7] In retrospect, sharing should have been expected since similar architectures permit common algorithms and data structures to be reused.

update various attributes of the unit when it is in use in the virtual memory system. Instance methods of the AddressTranslation class update this information during memory placement and replacement activities. In particular, the AddressTranslation class removeMapping method transfers reference and modification information for a block of physical memory from the dynamic address translation tables to the corresponding PhysicallyAddressableUnit instances.

5.2 Store

An instance of the *Store* class is a collection of all the PhysicallyAddressableUnit instances in the system. The *allocate* and *free* methods manage the assignment of *physical* memory to kernel subsystems including the virtual management and I/O subsystems. The allocate method is used to request a number of *bytes* of physical memory from the free physical memory pool. The request is satisfied by assigning the minimal number of PhysicallyAddressableUnit instances needed to hold that number of bytes. In the event that the allocate method cannot satisfy the request, it blocks. When the system replacement algorithm frees instances of the PhysicallyAddressableUnit class by swapping out the contents of infrequently accessed virtual memory to the backing store, it unblocks and fulfills unsatisfied requests.

5.3 PhysicalMemoryChain

Individual instances of PhysicallyAddressableUnit are not convenient data structures with which to program the manipulation of the arbitrarily connected physical memory regions that might correspond to a contiguous virtual address range or to a swap queue of pages. The *PhysicalMemoryChain* class defines a collection of PhysicallyAddressableUnit instances. PhysicalMemoryChains consist of a linked list of PhysicallyAddressableUnits, an aggregate size, and a byte offset into the first unit. These three entities are sufficient to describe an arbitrary resident virtual address region by its physical memory addresses. PhysicalMemoryChains are the primary form in which memory regions are passed between objects in the *Choices* virtual memory system. They describe memory regions for I/O and physical address mappings for AddressTranslations.

Constructor and destructor functions of the PhysicalMemoryChain class increment and decrement reference counts kept in the component PhysicallyAddressableUnits. The replacement algorithm will not swap the data held in a block to backing store if the reference count of its PhysicallyAddressableUnits instance is non-zero. This assures reliable I/O since the physical memory required for the I/O is first included in a PhysicalMemoryChain and then passed to the I/O subsystem.

6 The Logical Memory Layer

The *MemoryObject* class defines an abstract protocol for accessing logical memory. Logical memory is a sequence of identically sized indexed storage units. A unit size of one byte models a byte stream, while a unit size of 512, 1024, 2048, or 4096 bytes can be used to represent blocks on a disk device.[8] Instances of MemoryObject subclasses represent program instruction segments, stacks, disks, heaps, data spaces, and files.

The methods *numberOfUnits* and *unitSize* return the number and the size of the MemoryObject units. The methods *read* and *write* take an index as an argument and access the corresponding unit within the MemoryObject. The *offsetToUnit* and *unitToOffset* methods convert byte offsets into unit indices and vice versa. In addition to the index argument, the read and write methods also have arguments to specify the number of units to be read or written and a PhysicalMemoryChain instance. The chain provides the locations of physical memory blocks that are to be used in the read or write operation. Reading and writing physical addresses provides a virtual memory mapping-independent mechanism for I/O.[9]

The PhysicalMemoryChain argument to read and write requires further explanation as it would seem more convenient to use an argument specifying a contiguous range of virtual memory. The latter approach requires a virtual memory to physical memory translation of the argument inside the MemoryObject. This restricts implementations of the MemoryObject. An instance must either be mapped into the same virtual memory as the argument or it must be able to invoke a mechanism to translate a virtual memory address into a physical memory address using the virtual memory mapping tables that were used to construct the argument. For example, it is convenient to place the lowest level MemoryObject instances that drive devices like the disk in kernel space and execute them in supervisor mode. If address arguments were virtual address ranges, the abstract MemoryObject protocol would require read and write to have an additional argument that specifies the virtual memory with which to perform the translation. An instance of

[8] For the sake of implementation simplicity, the size of a unit is always a integer power of two

[9] Most I/O devices perform I/O using physical memory addresses, not virtual memory addresses.

the PhysicalMemoryChain class is used as the argument rather than an instance of PhysicallyAddressableUnit in order to promote locality when the instance of the MemoryObject represents a physical device like a disk. The PhysicalMemoryChain represents a contiguous logical entity in virtual memory but it may include disjoint blocks of physical memory. Were these to be read or written individually, the MemoryObject could not take advantage of their logical contiguity to read or write them efficiently to the device in a single request using scatter gather direct memory access hardware.

For convenience, the PhysicalMemoryChain constructor function builds the list of PhysicallyAddressableUnits from arguments that specify a starting virtual address, a length, and the virtual memory in which the address range is valid. As it builds the list, the function invokes other virtual memory system methods to make sure that the virtual memory specified in the arguments is resident and locked into physical memory.

Although the primary interface to a piece of logical memory is through its MemoryObject's read and write methods, logical memory may also be mapped directly into the virtual address space of a process. A virtual address mapped logical memory is directly accessible via the processor's instructions. Examples of such MemoryObjects includes those that represent the instructions or data of an executing application.

Unlike many virtual memory implementations which use one backing store for the whole of virtual memory, the *Choices* virtual memory scheme backs a virtual memory mapped logical memory to its MemoryObject. This allows *Choices* memory placement and replacement polices to take advantage of the characteristics of a logical memory. For example, memory references to a logical memory holding a stack will exhibit strong spatial locality. The replacement algorithm can simply discard memory furthest from the top of the stack yielding performance equal to a least recently used memory replacement algorithm without requiring the collection of access pattern information.

6.1 MemoryObject Subclasses

Subclasses of MemoryObject, shown in Figure ??, define different implementations of the MemoryObject abstract protocol. Current MemoryObject subclasses have been implemented to represent physical disks, disk partitions, sub-ranges of other MemoryObjects, Berkeley UNIX inodes, System V UNIX inodes, MS-DOS files and other new and experimental file system structures [?, ?]. These subclasses also exemplify the benefits of object-oriented programming and class inheritance in operating system implementations. The implementors of many of these subclasses were not required to know the *Choices* virtual memory management system. The interchangeability of the instances of the subclasses simplified constructing the utilities, paging, device access, and stream-oriented file systems while providing a flexible interconnectivity that allows, for example, paging a logical memory to a UNIX file.

7 The Cached Logical Memory Layer

The logical memory described in the previous section can either be accessed through the virtual memory system, or directly from secondary storage. Direct access of logical memory stored on secondary storage such as a disk suffers from access latency and limited device throughput. The cached logical memory layer uses physical memory to cache memory objects that reside on low latency devices like disks or other networked computers. The layer does not change the functionality of MemoryObjects, rather it speeds up the access to logical memory by allowing its contents to be accessed directly by the instructions of the computer.

The class MemoryObjectCache provides abstract protocols and methods for mapping portions of a logical memory supported by any instance of the class MemoryObject into physical memory using the functions provided by the physical memory layer (see Figure ??). Subclasses of the MemoryObjectCache class provide specific schemes or dynamic *caching* techniques to map all, part, or none of the data in a logical memory into physical memory.

The *cache* method of an instance of the MemoryObjectCache class ensures that a region of the logical memory data, in the form of a starting unit and a number of units, is resident in physical memory. It returns the PhysicalMemoryChain describing the physical memory caching this region. If the region is not resident, the cache method first invokes the allocate method of its Store. This returns a PhysicalMemoryChain instance. Next, it invokes the MemoryObject instance's read method with this as an argument (see Section ??).

Periodically, the virtual memory replacement algorithm is run to maintain adequate free memory for future Store allocations. The algorithm invokes the *release* method on one or more MemoryObjectCaches to free physical memory. The *modified* method of an instance of a PhysicallyAddressableUnit indicates

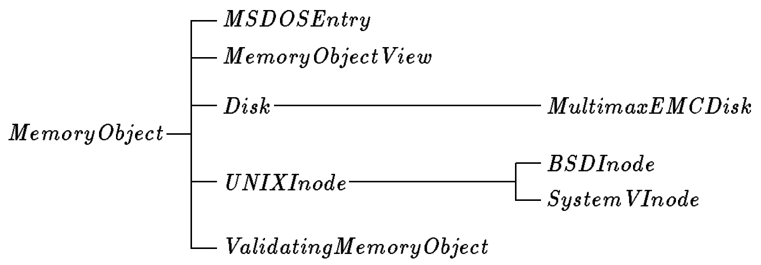

Figure 2: Some Classes in the *Choices* MemoryObject Class Hierarchy

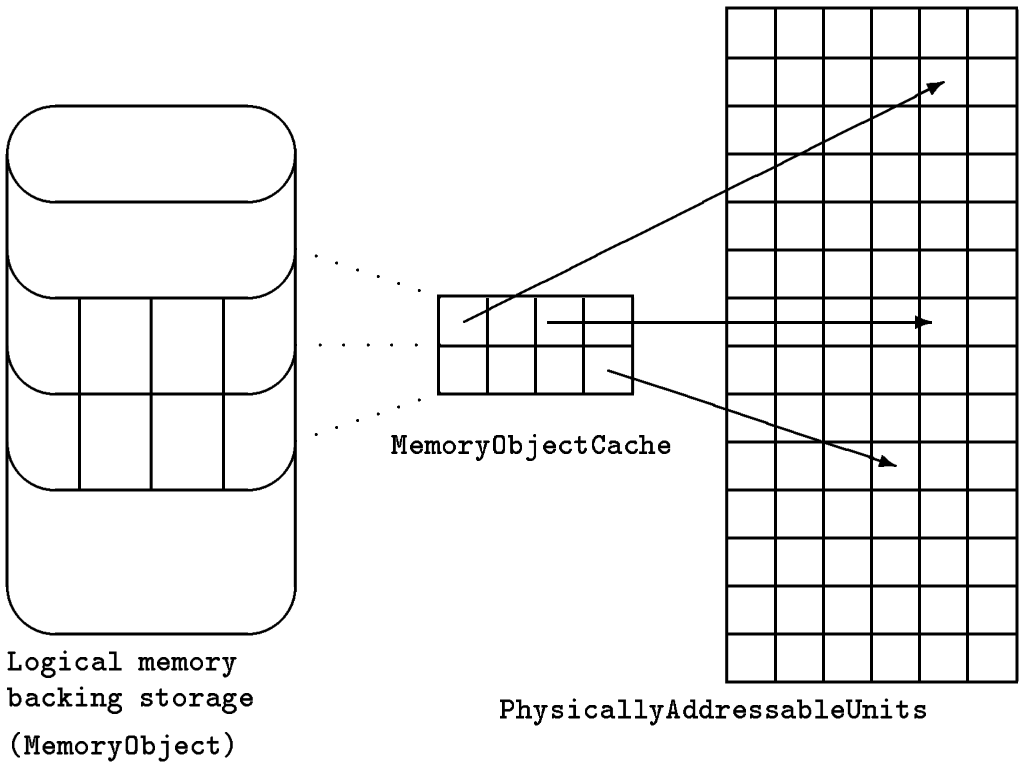

Figure 3: Relationship Between MemoryObjectCaches, MemoryObjects and PhysicallyAddressableUnits

whether a physical memory block has been modified since it was read from the cache's logical memory. The release method builds instances of the class PhysicalMemoryChain to represent all of the PhysicallyAddressableUnits containing modified data. The release method writes these instances to the cache's MemoryObject instance, ensuring the consistency of the logical memory in secondary storage. The method then removes the physical memory from the cache by returning an instance of PhysicalMemoryChain that includes all the freed physical memory blocks. *Choices* supports both global and cache specific virtual memory replacement algorithms. A global replacement algorithm incrementally updates all the instances of PhysicallyAddressableUnit that it manages in the system with usage information extracted from all the instances of the AddressTranslation class. It uses this information to implement replacement policies like *working set* or *least recently used*[?].

A cache specific replacement scheme is implemented by subclassing MemoryObjectCache and specializing the *replace* method. This method applies a cache specific replacement policy to the physical memory allocated to the instance of the MemoryObjectCache if the Store is below a threshold. It may be invoked within the cache or by a global algorithm initiated by the Store that frees physical memory.

This section outlines how logical memory is cached in physical memory. The complete details of the *Choices* virtual memory replacement mechanisms involve scheduling and are beyond the scope of this paper.

8 The Virtual Memory Layer

The previous sections describe the caching of logical memory into physical memory and the manipulation of the dynamic address translation hardware. This section presents the virtual memory layer. It unifies these mechanisms to provide virtual memory addressing of logical memory.

The *Domain* class provides the virtual memory layer abstraction. It maintains a collection of logical memories and their associated access rights together with a map of these logical memories into the virtual memory of an application (see Figure ??). Each logical memory is represented by an instance of MemoryObject cached in physical memory by an instance of MemoryObjectCache. When a logical memory is shared by more than one Domain, the cache is shared by the Domains. Each Domain has an associated AddressTranslation instance that it uses to maintain its virtual memory mappings.

The *add* method of a Domain binds a virtual address range to a logical memory. The add method is overloaded. One form binds logical memory to specified virtual memory addresses. The other form allocates a virtual address range sufficient to map the logical memory. Add also records the Domain binding in the instance of MemoryObject representing the logical memory using the *addToDomainList* method. The release method of a shared instance of MemoryObjectCache uses the Domain binding information in its MemoryObject to notify Domains that logical memory data has been removed from the cache.

The *remove* method deletes the mapping between virtual memory addresses and a logical memory. First, it invokes the AddressTranslation removeMapping method to invalidate any hardware physical address translation mappings. This also updates reference and modification information in those instances of PhysicallyAddressableUnit that correspond to the physical memory blocks storing resident data (see Section ??). Then, if the instance of MemoryObjectCache representing the logical memory is not referenced by any other Domain, it is deleted. Deleting the cache will force any modified resident data to the backing store. Lastly, all references to the logical memory in the Domain are removed.

The add method binds virtual memory addresses to logical memory. However, the logical memory must be cached in physical memory before it can be accessed by the processor. The *fixFault* method ensures that a logical memory block is cached in physical memory and updates the AddressTranslation with physical addresses from the cache. Its argument is a virtual address. First, it converts the virtual address into a logical memory reference and an offset pair. Next, fixFault invokes the cache method on the corresponding instance of MemoryObjectCache (see Section ??) using the offset and the size of the block as arguments. The cache method returns an instance of a PhysicalMemoryChain to represent the block of cached logical memory and its location in physical memory. Next, fixFault invokes the addMapping method on the Domain's instance of AddressTranslation (see Section ??). The arguments to the addMapping method include the PhysicalMemoryChain instance, the access rights associated with the logical memory in this Domain, and the virtual address of the logical memory block.

When a *Choices* process accesses a virtual address that generates a virtual memory hardware fault, the exception handler for that fault invokes the fixFault method of its Domain using the virtual address as an argument.

It is easy to share a logical memory between dif-

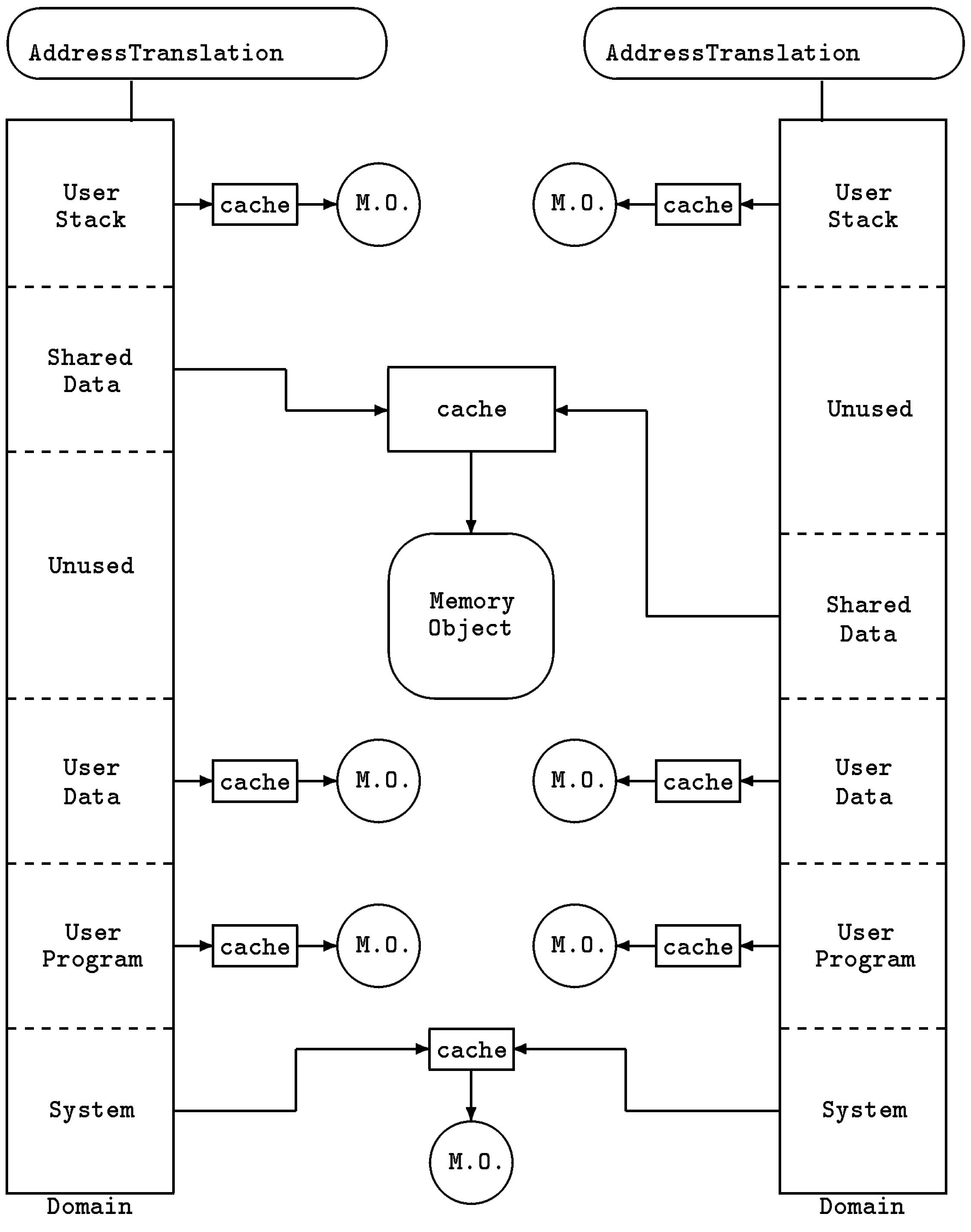

Figure 4: Relationships Between Objects Managing a Virtual Address Space

ferent instances of Domain. Because an instance of a MemoryObjectCache is independent of virtual memory, a logical memory can be mapped into different ranges of virtual address.[10] As the instances of Domain are responsible for assigning access rights, the logical memory can also have different access rights in different Domains.

A virtual memory mapped logical memory can be shared by different instances of Domain residing on different nodes of a distributed system. Each node sharing the logical memory has a local cache for the memory and uses a cache consistency protocol to keep its cache consistent with the other caches and the logical memory[?].

Processes may change the logical memories accessible from a Domain as part of a scheme that implements *inter-address-space* objects. When a method is invoked on the "proxy" of the inter-address-space object (an *ObjectProxy*), the Domain of the invoker is modified to reflect the memory objects that are encapsulated by that object. On exit from the method, the Domain is modified to remove those memory objects that are encapsulated by the object.

9 Conclusions

This paper describes the object-oriented design and implementation of the virtual memory and backing storage management system for the *Choices* operating system. The operating system is implemented in an object-oriented language (C++), consists of over 40,000 lines of code, and runs native on an Encore Multimax multiprocessor. C++ provides a very efficient implementation of method invocation and inheritance.

The memory management system is organized into an address translation layer, a physical memory layer, a logical memory layer, a cached logical memory layer and a virtual memory layer. Abstract classes define the interfaces to these layers. Concrete subclasses implement specializations of these layers on the Multimax and other hardware. We describe the design of the memory management system in terms of these classes and exemplify the object-oriented paradigm with specific implementation and design examples.

The object-oriented approach allowed us to prototype different virtual memory implementation schemes, reuse code, maintain machine-independent abstractions, encapsulate implementation decisions, separate policy from mechanism, and provide for well-defined interfaces.

[10]The logical memory must contain position independent data. Files and disks are good examples.

In *Choices*, the virtual memory concepts extend the principles of operating design in the direction of an object-oriented design and implementation. In particular, the class hierarchical representation of these principles provides an organization for managing a variety of operating system implementations. The use of inheritance is important in reducing the complexity of the operating system design and implementation.

Our operating system design includes:

- a virtual memory that is composed of independent logical memories mapped into the address space.
- shared logical memories both within and between virtual memory address spaces.
- logical memories mapped onto multiple, arbitrary virtual memory address ranges.
- independent backing stores for each logical memory.
- a choice of different backing stores.
- alternate logical memory object access through a file-like read/write access protocol.
- a physical memory copying function based on PhysicalMemoryChains that provides an abstraction for exploiting scatter gather direct memory access hardware.
- a choice of local page placement and replacement algorithms for individual cached logical memories.
- a framework of abstractions and reusable software that can be used to build experimental virtual memory systems.

Choices is currently being ported from the Encore Multimax to an Intel iPSC/2 hypercube. The memory management system also supports shared networked virtual memory. An account of the *Choices* virtual memory system extensions for networked virtual memory is being written. Although many of the applications of the *Choices* virtual memory management scheme have yet to be explored, we believe the class hierarchical, object-oriented approach we have adopted will allow us to attack them in a rigorous sequence of steps.

References

[1] P. J. Denning A. V. Aho and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, January 1971.

[2] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[3] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[4] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[5] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[6] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, pages 71–79, September 1987.

[7] Intel Corporation, Santa Clara, California. *80386 System Software Writer's Guide*, 1987.

[8] Gary M. Johnston. *Data and Process Migration in Distributed Virtual Memory Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989 (forthcoming).

[9] Peter Madany, Douglas Leyens, Vincent Russo, and Roy Campbell. A C++ class hierarchy for building UNIX-like file systems. In *Proceedings of the USENIX C++ Conference*, October 1988. Also Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

[10] Peter W. Madany, Roy Campbell, Vincent Russo, and Douglas E. Leyens. A class hierarchy for building stream-oriented file systems. In *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, 1989.

[11] National Semiconductor Corporation, Santa Clara, California. *Series 32000 Databook*, 1986.

[12] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1985.

[13] Richard Rashid et al. Machine–independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[14] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1981.

[15] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1988. Also Technical Report No. UIUCDCS–R–88–1415, Department of Computer Science, University of Illinois at Urbana-Champaign.

[16] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[17] L. D. Wittie and A. Van Tilborg. MICROS – a distributed operating system for MICRONET – a reconfigurable network computer. In H. A. Freeman and K. J. Thurber, editors, *Tutorial: Microcomputer Networks*, pages 138–147. IEEE Press, 1981.