The NanOS Object Oriented Microkernel: An Overview

F.D.Muñoz-Escoí J.M.Bernabéu-Aubán

Technical Report ITI-ITE-98/3

Abstract

NanOS is an object-oriented microkernel. It provides support for objects, agents and tasks and a basic service: object invocation. Upper levels of software must be built as a set of objects residing in one or more agents.

An agent is a protection domain. Each agent has its own address space and provides access to some remote objects using object descriptors. The tasks are the units of execution. Tasks may change the agent where they are being executed if they invoke objects which do not reside in its own protection domain.

This microkernel also offers some support for task scheduling and memory management. The scheduler has been placed initially in the microkernel, although in future releases it may be placed outside. Memory management uses memory representatives which are related to external pagers and cache objects. The pager is able to provide the memory when it is needed by a local cache. A cache object is the holder of parts of the memory object in a given node.

1 Introduction

Current trends in operating system design are likely to adopt a minimal kernel approach, either based on microkernels [1, 2, 6, 11, 9, 12], cache-kernels [3] or exokernels [5]. All of them offer support to develop part of a traditional operating system at user level, making possible to download later part of this software at the kernel level. Exokernels are the most extreme solution, since they do not provide any high level abstraction and they only multiplex some computer resources (CPU, memory, devices, etc). On the other hand, microkernels provide a set of abstractions (objects, execution threads, protection domains, synchronization tools and others) which facilitate the construction of upper level software modules.

Our microkernel offers as its main abstraction the *object*. Thus, user level software is constructed as a collection of objects. To assist in the management of objects, it also provides *agents* as protection domains where objects have to be placed before their methods are executed. The dynamic abstraction is the *task* or execution thread which may visit several agents as a result of consecutive object invocations.

All these abstractions offered by the microkernel are related via the *object invocation* service, which is the most critical service provided. Since several parts of a traditional kernel are now placed at user level and decomposed in a set of objects, communication among them will be common and the performance of the operating system will depend on the performance of the interdomain communication service; i.e., object invocation in our case.

Our object invocation service does not require context switches and uses stack pre-allocation. So, when a task invokes an object's method it is immediately transferred to the target agent's entry point, where it takes another stack and finds the method to be executed.

This microkernel offers good services for local interdomain communication. Interconnecting several machines with a private network and extending the object invocation services with an ORB [8] we have a good basis to develop a powerful distributed operating system. The target system to build using NanOS will manage a cluster of machines offering a single-system image. Moreover, the ORB will be extended to support replicated objects, offering a good basis to develop highly available applications.

The rest of the paper is organized as follows. Section 2 describes the system design goals. Then, three sections explain the general architecture concepts of the system. Section 3 describes the memory abstractions, grouping them around the virtual space object. Section 4 explains the object concept and some items related to it, mainly the capabilities or object descriptors. The sections about system architecture end introducing the dynamic abstractions. As such can be considered the task, thread and agent objects.

Section 6 describes the kernel organization. In that section the kernel is viewed as a set of components, and the relation between abstractions and kernel software components is explained, too. Finally, the **Summary** section includes some conclusions about the current release and outlines future extensions.

2 System Overview

This section describes the services provided by our kernel, how the system uses them and which were the kernel design objectives.

2.1 Kernel Services and Abstractions

Our kernel provides to the rest of the system a set of abstractions, which can be used and combined to offer different operating systems views. These basic elements can be divided into two main classes: the static abstractions and the dynamic ones.

2.1.1 Static Abstractions

Static abstractions refer to what is executed in the system. Since NanOS is an object oriented operating system, the object is its main static abstraction. But many other abstractions of this kind are needed to support the object concept. They are outlined in the following lines:

- **Object.** An object maintains a set of data (object state) and offers a set of operations to manipulate these data (object interface.) In our system, each application or system component consists of a set of objects of this kind, whose interfaces can be invoked, leading to the execution of their methods and the update of their states.
- **Virtual Space.** Since the objects must be located somewhere, we provide virtual address spaces to place them. Each virtual space provides a protection boundary to the objects residing in it. Also, virtual memory techniques can be applied to provide its memory.
- **Memory.** Each memory object represents a chunk of the system memory. Using them, different ranges of addresses in the virtual spaces can obtain physical memory and memory sharing among virtual spaces is possible, too.
- **External Pagers.** To apply virtual memory techniques to the address spaces we need mapping and paging operations to load or unload different parts of the physical memory into the spaces; external pager objects provide these operations in our system.

Using these four abstractions, we provide the means to define the static view of an application. So, we know how many objects build up our application and how they are distributed in our system memory. But something lacks, we can not execute any piece of code belonging to the application without the dynamic abstractions.

Figure 1 depicts an example of some static abstractions of NanOS, and how they may be combined to provide the static view of a multiprocess system.

2.1.2 Dynamic Abstractions

Dynamic abstractions are related to the elements which carry out the execution of programs. In our system, the following dynamic abstractions can be distinguished:

- **Task.** A task is the system abstraction which represents a program execution. So, to unambiguously describe a task, the kernel needs to know the contents of the CPU registers and a link to the code it was executing.
- **Agent.** An agent is the abstraction that allows the relationship between the pure dynamic abstraction (the task) and the main static one (the object.) The agent is basically a protection domain. It encompasses a virtual space and a set of references to the objects accessible from this virtual space.

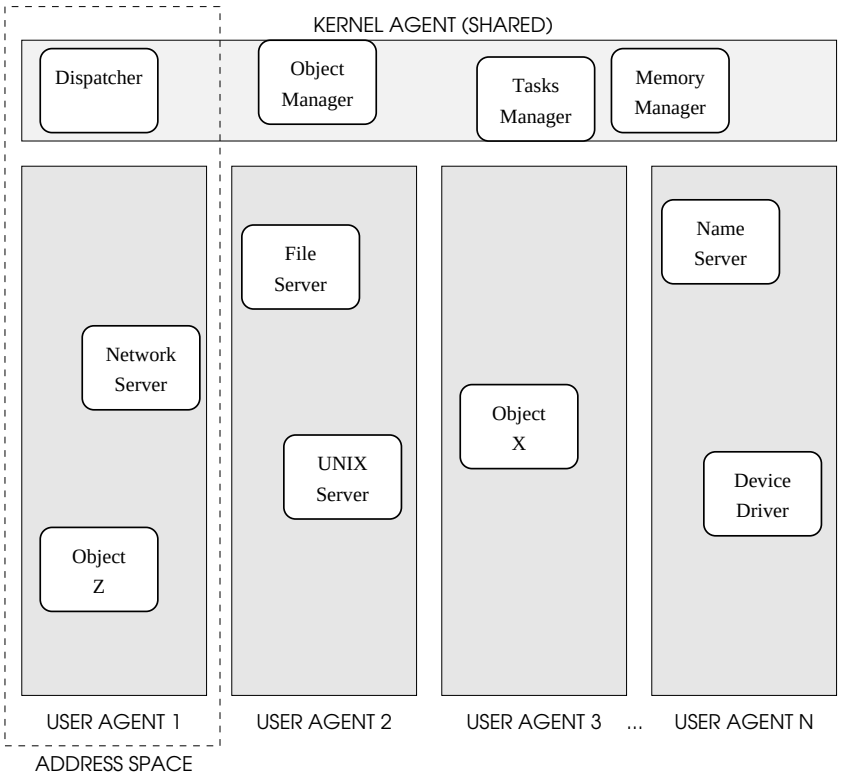

Figure 1: Virtual spaces and objects in a system node.

In our case, the execution of a task is not constrained to a unique agent. So, a task can sequentially invoke and execute methods belonging to objects placed in different agents. Also, several tasks can be executing code in an agent at a time.

To complete the picture, the services provided by the kernel are sketched.

2.1.3 Kernel Services

Apart from providing and managing all abstractions shown above, the kernel is concerned in providing a very important service to any task. This is the **invocation service**. Using it, a task can invoke an object placed in another agent or even in the kernel.

The invocation service is the most important part of a microkernel, since the performance of any service not present in a microkernel depends on it to be carried out with efficiency and sufficient performance.

Besides providing invocation facilities, our kernel also provides some memory management, tasks scheduling, tasks synchronization and device drivers support.

2.2 Design Objectives

In addition to the common objectives of a kernel design – namely good performance, efficiency and clean interfaces – other design goals have been considered:

- **Extensibility.**

  This kernel adopts the micro-kernel philosophy. Thus, only the essential services can be found initially in supervisor code. But this minimal approach has often led to performance decreases. A solution to this problem is to provide mechanisms that allow the inclusion of new objects into the nucleus code at run or boot time.

Increasing the kernel functionality in this way allows us to test thoroughly the servers to be added. These tests may be done at user level before doing the kernel extension. Since the server to be tested is a user object, nothing prevents us from testing simultaneously several implementations of that server. Only the instance that best matches the system needs will be finally included into the kernel.

- **Orthogonality: Kernel as an Agent.**

  The kernel must manage itself as any other agent in the system. If this is possible, the kernel Memory Manager object will be able to provide mapping and swapping services to regions of the proper kernel address space and the kernel tasks will access objects placed in other agents using the generic invocation service.

  The kernel presents some peculiarities to be considered. For instance, its address space has a different range of addresses and must be always active. But managing it as another agent will ease some common tasks, as making calls to user code (up-calls) and adding new code elements to its address space — using `Map()` calls to its `VirtualSpace` object.

- **Invocation transparency.**

  The way an application programmer sees an object invocation will be always the same, independently of the object location. There are three possible object locations to be considered: the same address space, another address space in the same node and some address space in another node.

  In the first case, for performance considerations, the invocation should be done using the static linking mechanisms provided by the implementation programming language. When the invocation involves two objects in the same node but located in different address spaces, the invocation service provided by the local kernel will suffice. The latter case needs the cooperation of the two kernels and some additional objects as net drivers and net servers. Currently, we are designing an ORB which will offer this kind of service.

  Invocation transparency can be achieved using stub objects in all three cases. The internal machinery of stub objects in the two non-trivial cases has to know about the object descriptors that every kernel manages.

  Object descriptors introduce a level of indirection when an object invocation is done. Using them, user applications do not require modification when the target object is migrated. The migration is registered in the kernel and does not affect the user code. Only the descriptor-location relationship is updated.

- **Portability.**

  The kernel has to be designed in a way that isolates the machine-dependent details in a minimum set of objects. These machine-dependent objects will have a machine-independent interface that can be used by the rest of the objects that build up the kernel.

  Porting the system to another architecture will be as easy as rewriting the machine-dependent objects considering the target machine features. The current version of our kernel has only a few objects of this kind. They refer to the following hardware elements:

  CPU. The object associated to this hardware item will manage the masking of the CPU interrupt levels and the interrupt enabling / disabling.

  CPU Context. A class is used to model the set of general purpose registers. It offers methods to get a copy of the actual registers into the object state and to restore these registers from the current object state. Objects of this kind are used in task switching.

  MMU. An object will offer methods to change the MMU context (virtual space) and to update the virtual to physical translation tables.

  Besides the objects presented above, some assembler code is required to initialize the machine at boot time and to call the high level code. Despite this, the current kernel version has less than a 25% of machine-dependent source code.

  Our initial design has been already ported to different hardware architectures. Nowadays we have a Sun SPARC and a PC releases of our microkernel. The first release was implemented on Sun SPARCstations and was ported recently to the other one. The original design was successful at this level, since no modification of the machine-independent code was needed.

3 Memory Management

The memory management provided by the kernel is based on three kinds of public objects. They are the memory object, the pager and the virtual space. These objects and their relationship are similar to the ones described in [7].

3.1 Memory Objects

The memory objects model the system physical memory. They may be either named or anonymous.

Named memory objects are the system support to memory that is backed in secondary store. Files and other persistent objects are structured as memory objects of this kind.

Anonymous memory objects are the system view of some main memory areas that can be requested by the agents to back transient blocks of memory. Task stacks and application heaps are two examples of memory objects that do not need a name.

As stated above, a memory object is the representative of a piece of the system memory. These objects will be used to build the virtual address spaces. An address space receives its memory from the mappings of memory objects at certain virtual address ranges.

Building the virtual address spaces in this way will ease the memory sharing among several address spaces, even if they reside in different nodes. Memory sharing is explained in more detail in section 3.5.

3.2 Pagers

The second class of public object related to memory management is the pager object. Memory objects do not offer paging operations. They are reserved to external pager objects which will be linked to their paged memory objects.

External pagers have the following advantages over self-paging memory objects:

- The actual memory that backs the memory object is not forced to reside in the same node where the memory object is being mapped. The location of this memory depends on the location of the pager object. The same pager object can be associated to different *instances* of a memory object mapped in different nodes.
- Pagers are a good place to locate memory coherence policies. These policies may differ from pager to pager.
- It is possible to replace the behavior of the paging operations without any modification of the memory object. A link to other pager object at bind time will suffice.

3.3 Virtual Spaces

Most of the present workstations have a memory management unit (MMU) that allows the isolation of user applications in different address spaces and gives support to virtual memory techniques. Our kernel provides the `VirtualSpace` object, which comprises both concepts.

A typical address space is structured in the following way in our system:

- **User space.**

  The user space area is placed in the lower addresses of each virtual space. It uses the addresses from 0 to 3 Gb in the current PC version.

  The organization of this range of addresses is almost free. The system compels to map a code memory object at a well known low address. This memory object will store the code that serves the requests routed to objects placed in the rest of the user space. Other memory objects backing the objects' state (data) or the tasks' stacks can be placed anywhere.

- **Kernel space.**

  The kernel is mapped at the highest addresses of each address space. Therefore, the memory allocated to the kernel is shared by all the virtual address spaces and the kernel is always mapped and active.

A `VirtualSpace` object is provided by the Memory Manager to represent one of these address spaces. The memory of an address space may be provided using the `Map()` method, which maps the memory to a given range of the address space.

Mappings are registered in the `VirtualSpace` object using two auxiliary objects: the `Region` and the `Cache`. A `Region` keeps the association between an address range of a virtual space and the section of the memory object that backs these addresses. The `VirtualSpace` uses the regions to control its ranges of used and free addresses.

The `Cache` object is the image of a given memory object in a system node. As `Region`s register mappings in terms of virtual addresses, `Cache`s associate memory objects with a series of physical memory pages. Thus, the `Cache-Region` relationship determines the virtual to physical mapping that has to be implemented by the MMU.

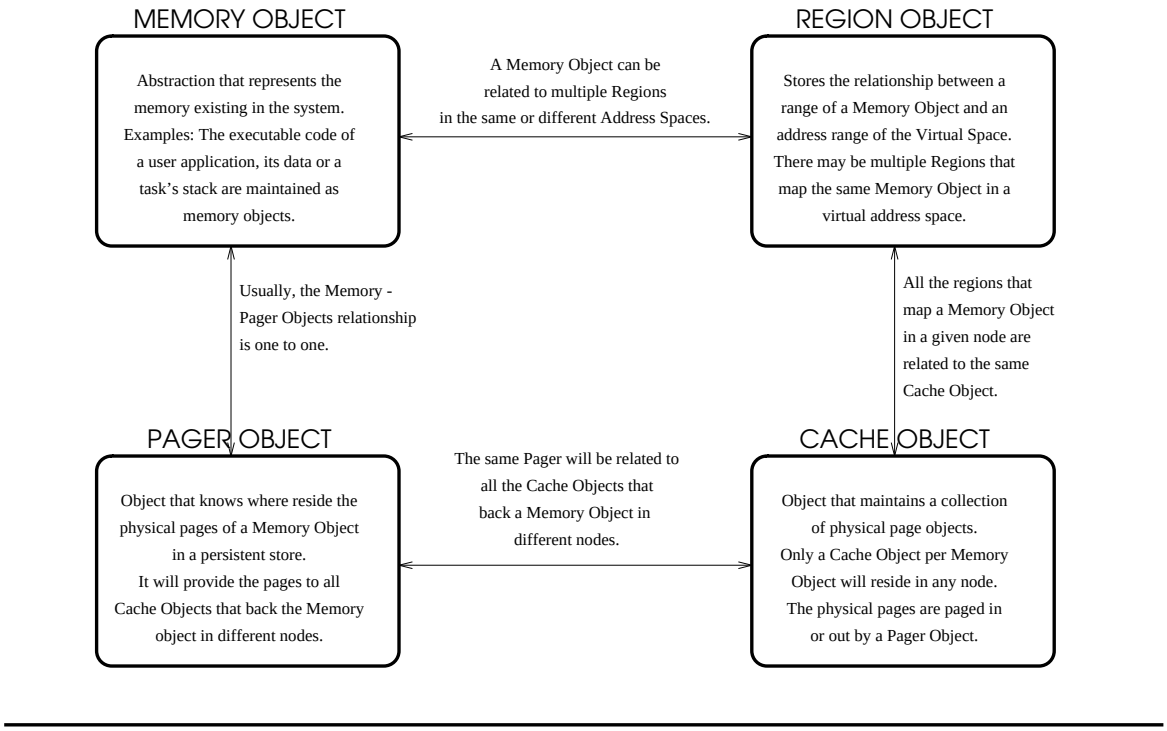

Figure 2: Relationship among different objects after a mapping operation.

3.4 Mapping Memory

To map memory to an address range of a virtual space, the kernel Memory Manager needs the following set of objects:

- A **Memory Object** that is the representative of the memory to be mapped. We assume that this object has been previously obtained by the mapper task.
- A **Region Object** that maintains the range of virtual addresses where the memory object has to be mapped. It will also store a reference to the `Cache` object that has the physical pages which builds up the local copy of the memory.

  The Memory Manager must check that the range of addresses where the memory object has to be mapped does not overlap with any other used Region. Otherwise, the `Map()` operation will be rejected.
- A **Cache Object**. As it was previously stated, a Cache maintains a set of physical pages that have the local image of the memory being mapped.

A local Cache object backing the MemoryObject must exist in the node where the Map() operation has to be done. Otherwise, the Cache has to be created in the first phase of the map request and it needs to be bound with a Pager object that knows about the physical location of the memory represented by the MemoryObject.

- An **MMU Object** which offers a set of methods that allow the installation of the physical pages maintained by the Cache object into the target virtual space.
- Some **Frame Objects** that build the cache memory.

Figure 2 shows the relationships established between some of the objects quoted above. The Frame objects are used by the Cache, which maintains a list with all the physical pages currently in main memory that belong to the cache.

On the other hand, the MMU object is required when a page fault raises. In the page fault processing, the physical page is requested to the Pager object and it is installed in main memory using the methods provided by this MMU object. The Frame object representing the memory recently installed is also included in the list maintained by the Cache object.

3.5 Sharing Memory

Figure 3 shows how some memory is shared between two address spaces after the same memory object has been mapped to both of them. While the mapping is being done, a Region object is created and associated to the VirtualSpace object. This reflects that the address range is not free and it also determines which Cache object will provide the physical memory represented by the memory object recently mapped.

If the same memory object is mapped to another virtual address space (or to another range of the same address space), a new Region object will be created but the Cache object is reused. The Map() method of every VirtualSpace object is smart enough to locate the appropriate Cache object in its node and attach its Region object to it. If no cache exists for the mapped memory object a new one is created.

An external object, the MemObjMgr maintains the relationship between MemoryObjects and Caches in every node. Thus, the VirtualSpace object can ask it for the existence of a cache backing the memory object in its node.

The Cache object is able to maintain the whole memory represented by a MemoryObject independently of the memory object ranges that have been mapped. In fact, only those pages that have been referenced by any task accessing the mapped addresses are kept in the cache. The other pages will never be installed. The addresses where the memory object has been mapped and which portion of the memory object was used in the mapping are maintained in the Region object and they change among successive Map() operations.

4 Objects

An object consists of a set of data — known as the object's state — and a set of operations, referred to as methods, which constitute the object interface and which are the only way to access the object's state.

Our kernel is structured as a set of objects of this kind that provide a set of interfaces and a basic invocation service to the rest of the system. Using the invocation service, a user task may request operations provided by the kernel objects or by objects that reside in other domains.

Kernel objects provide methods that allow user tasks to register new user objects in the system, delete previous registrations, get object references, etc. User level applications must use these facilities to organize themselves as a collection of objects that will interact among them using the IPC mechanisms provided by the kernel (namely, the invocation service and the memory sharing).

The rest of this section explains the operations provided by some kernel objects related to the life-cycle of an object.

4.1 Object Registration

Once an object has been created, it must be registered in its node's kernel. Before an object registration, no invocation can be directed to it since the kernel does not know anything about it.

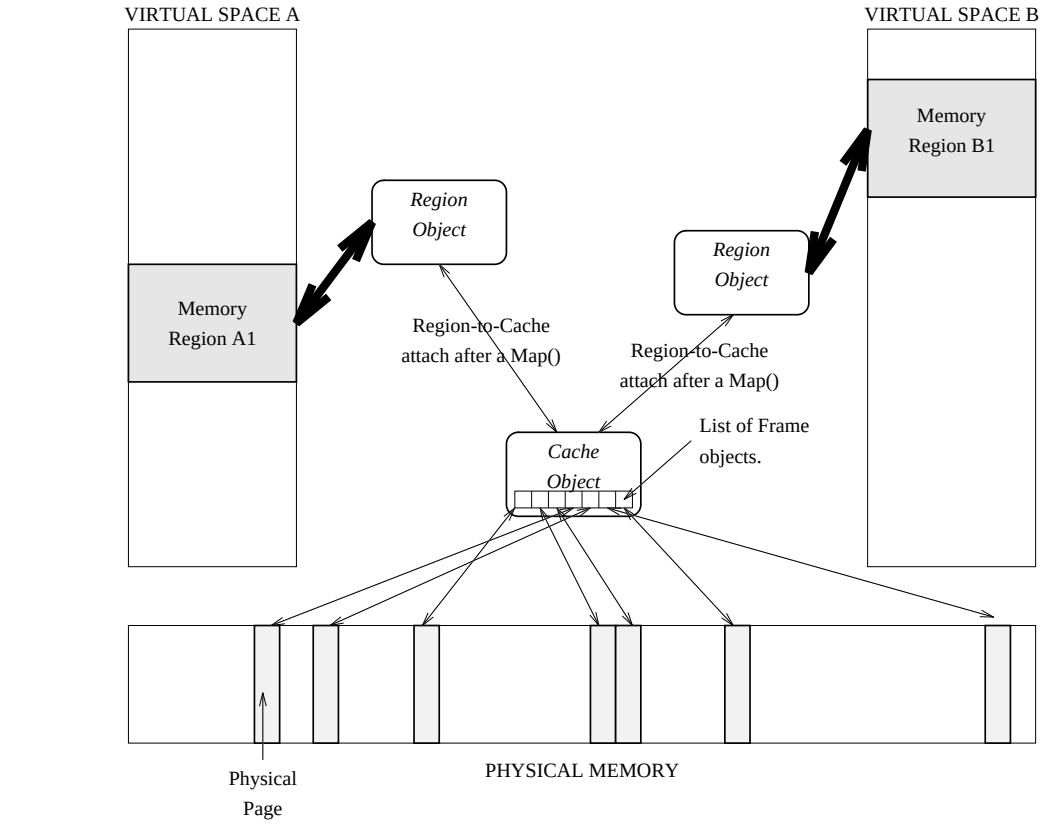

Figure 3: Memory shared between two address spaces in the same node.

To register a new object, the `ObjectMgr`'s `Register()` method can be used. The location of the object is given in the input arguments of the method, and a valid descriptor to it is returned in the output one.

The kernel maintains a table of object locations, associating each object to an internal object identifier. Besides this table, additional descriptor tables are also maintained. Each table of this kind relates a set of descriptors, belonging to a task or an agent, with internal object identifiers.

Object locations and object descriptors are the only information stored in the kernel about an object. The object location is the internal data used by the kernel to identify the agent where the object resides. The descriptor is the user level identifier for an object; the invoker task uses it to tell the kernel which object is invoking.

As a consequence of this design, the kernel neither controls the user-level object interfaces nor their object states. Both of them can be structured and managed freely by the user code.

4.2 Object Invocation

The object invocation service is requested using a software trap. Some assembler code is required to direct the caller task to the `CallMgr` object. This object checks the received arguments, especially the object descriptor. If the descriptor is invalid, the invocation is aborted.

Since descriptors are resources that belong to agents and tasks, either the `AgentMgr` or the `TasksMgr` are needed to locate the target object. Moreover, the `TasksMgr` is always used to spawn an additional thread which will execute the code of the invoked object. In Section 5.1 the steps followed in an object invocation are explained in detail.

4.3 Descriptor Handling

Descriptors are grouped in tables. Each descriptor table belongs to a different agent and it is represented by an `AccessTable` kernel object. An `AccessTable` object relates the object descriptors to global kernel identifiers. Object descriptors have only sense if they are dereferenced using the `AccessTable` where they reside.

`AccessTable` entries can only be filled by the kernel. This prevents object descriptors to be forged at user level.

The kernel provides the following operations related to object descriptors:

- Automatic generation of the first descriptor when the object is created. This descriptor is usually assigned to the agent where the object has been created.
- Creation of a new object descriptor as a result of a `LetAccess()` call to the `AgentMgr`. Using this method, an agent gives one of its descriptors to a second agent.
- Creation of new object descriptors as a result of an object invocation. The kernel provides a facility to translate object descriptors in invocations. When an object is being passed as an argument in an inter-domain call, the caller can indicate to the kernel that the descriptor must be converted to the appropriate descriptor value in the target agent.
- Destruction of a given descriptor, only if the requester of the operation owns the descriptor to be removed.

4.4 Reference Counting

A reference count is maintained for each object that has been registered in the kernel. Although a `Delete()` method is present in the `ObjectMgr` interface to explicitly delete the registration of a user object, the reference count is provided to automatically unregister an object that is not being referenced by any task or agent.

The reference count is increased when an object descriptor is created by the kernel. The count is decreased when a descriptor of the object is removed. If it reaches a value of zero, the object registration is deleted and the object is no longer accessible.

Descriptors are removed automatically when they belong to a task or agent that is being deleted.

5 Threads, Tasks and Agents

Threads, Tasks and Agents are the three abstractions related to the dynamic view of the system:

Task. It models a thread of execution. During the execution of an application some calls to objects located in other address spaces may be needed. The task abstraction supports these inter-domain jumps and returns. This implies that a task switch is not required when the same thread of execution traverses different domains.

Thread. A thread is the view of a task into an address space. Threads maintain information about where has been mapped the task's stack. Also, once a task has left the address space as the result of an object invocation, the kernel stores into a `Thread` object information that will allow the restart of the task after the invocation return.

The life of a thread commences when a task calls an object placed in its address space. The thread is maintained until the invocation that started it finishes and the Task returns to its previous address space.

A thread can be in two states: **Active** or **Passive**. Only one thread per task is in the active state: the last one. The others are in the passive state, and will be reactivated when the younger task threads have been terminated.

Agent. The agent is the system entity that maintains some objects and isolates them from the rest of the system — it is a protection domain. It includes an address space where objects reside. An agent also maintains some object descriptors, allowing the tasks that run in it to invoke objects placed in other agents.

Besides providing the tools to initiate an object invocation (acting as a client), the agent also supports the entry point concept (now considering the agent as a server). In our system, there is one entry point per agent. Some code must be provided to route the entry calls to their target objects.

In the the rest of the section the invocation process and the protection provided by the object descriptors are analyzed in more detail.

5.1 Tasks and Object Invocations

A brief explanation of an object invocation will clarify the relationships among tasks, threads and agents. The following points define a common scene previous to an object invocation:

- A task is executing code of an object in its address space.
- This code contains a call to another object that is not placed in this address space.
- The object to be invoked is referenced by an object descriptor. This object descriptor usually is a resource owned by the agent.

To do the invocation, these steps must be followed:

1. The invoking task makes a call to a client stub, passing the invocation arguments.
2. The client stub raises a trap that is attended and processed by the kernel.
3. The kernel gets the object descriptor and redirects the invoking task to the agent where this object resides.
4. To do that, the kernel creates another thread and associates it to the invoking task. Thus, the task can execute code placed in another agent.

As we can see, an agent provides a set of accessible objects to a task. If the task decides to invoke one of these objects, a new thread will be created. After its creation, the task is placed in the target agent and executes the code of the invoked object. Now, it has access to a possible different set of objects depending on the set of object descriptors owned by the new agent.

As stated previously, our threads are a very light abstraction. In fact, a thread data structure only maintains the stack pointer and a return address. So, linking a thread to an invoking task does not need a task switching and can be done in a few microseconds.

5.2 Protection and Object Descriptors

Object descriptors provide a means to access objects placed in different address spaces. They are created by the kernel and maintained in descriptor tables. The descriptor value can be interpreted by the kernel, who finds the correct table entry and gets the identifier of the object referenced by the descriptor. Once the object identifier has been obtained it is easy to find the location of the object and to transport the invoking task to the entry point of the target object.

If an object descriptor is forged at user level and used to invoke a random object, the kernel will detect it because it will point to an empty table entry. The invocation is aborted and the task cannot gain access to unauthorized objects.

If the forged object descriptor references a full table entry no violation is made: the task already had access to this object. Moreover, a method number is also required to do an object invocation and they may be very sparse. Therefore it is very difficult that a randomly generated object descriptor and method number pair could result in a valid invocation.

6 Kernel Organization

The kernel not only provides the object abstraction to the upper levels. It is also organized as a collection of objects highly related, each one providing a well defined set of services to other kernel objects or to user level ones.

6.1 Kernel Objects

The most important kernel objects are:

AgentMgr. This object manages the agents of its node. It receives requests to create new agent objects and requests to delete any of the current ones. It also provides a little management of the agent descriptor tables, allowing the registering, deletion and copying of object descriptors.

CallMgr. The `CallMgr` object serves the kernel entry point. Since the kernel only provides the invocation service, this object has to combine the services provided by other kernel objects to route the invoker tasks to their target objects.

Usually, the functions carried out by this object are:

- To locate the target object and its entry point. An object descriptor is needed to do that.
- To create new threads to execute the invoked code.

Dispatcher. The `Dispatcher` object manages the task switches. It maintains the CPU context of each task and also provides methods to create, delete and get task identifiers at low level — the `TasksMgr` object is built on top of it.

IntrMgr. The `IntrMgr` (interrupt and trap manager) object allows other kernel objects to request to be notified of a given trap type arrival. The objects that require this kind of service are referred to as **interrupt handlers**. The `Clock` is a good example of an interrupt handler.

The `IntrMgr` not only manages the service of external interrupts. It can also notify the arrival of software traps and other exceptions.

MemoryMgr. The memory manager object groups the functionality provided by other kernel objects related to memory abstractions. They are the following ones:

- **Virtual Spaces Manager.**
  As a virtual spaces manager, the `MemoryMgr` serves requests to allocate, map, unmap and free different address ranges of a virtual space. It will also manage the creation and deletion of virtual spaces.
- **Frames Manager.**
  When the memory manager acts as a manager of physical pages, it has to maintain a set of `Frame` objects that comprise the physical memory of the machine where it is running. This set of objects is built at initialization time, once the amount of physical memory has been found out.
  Later, the manager maintains information about which pages are free and updates the reference count of the used ones.
- **Caches Manager.**
  In the role of caches manager, the `MemoryMgr` associates some `Frame` objects to its owned caches and relates the `Cache` object to the `Memory Object` and `Pager` that know about the location of its memory in secondary storage.

As we can see, some additional objects are required to provide the whole memory management. Part of them are used directly to implement the three objects seen above or can be thought as independent objects, also at kernel level. Examples are the `Region`, `Frame`, `MMU` and `PageTable` objects.

But another part may be found at user level: `MemObjMgr`, `Pager`s, etc. These ones can be easily replaced. So, some policies as cache coherence might be modified if the `Pager` objects that provide them are updated.

ObjectMgr. The object manager allows the registration of new objects in the system node, deletion of previously registered ones, locating an object given a valid descriptor and so on. This manager keeps the location of all the objects registered, assigns kernel identifiers to them and manages a reference count for every registered object.

Reference count handling allows the deletion of an object registration when all of its references (usually, object descriptors) have disappeared.

Scheduler. The `Scheduler` object maintains a set of queues where all the ready tasks are waiting for its next CPU time. It provides methods to include new tasks in the ready queues or to suspend a ready task.

The scheduling policy depends on the implementation of this object. The current instance deals with 32 different priority classes, each one managed using a Round-Robin strategy.

Since the interface of this object has a few methods, the replacement of the scheduling strategy is not difficult.

SynchroMgr. The `SynchroMgr` object controls the synchronization objects. There are three kinds of them, namely semaphores (`Semaphore` class), locks (`Lock` class) and event variables (`EventVar` class). They are registered and handled as usual objects. Therefore, descriptors of them can be transferred between different agents and they can synchronize tasks running in these agents or in the kernel.

A `Semaphore` object provides the `P()` (test) and `V()` (increment) methods. They follow the behavior of the original semaphore primitive [4].

The `Lock` object is a variant of the previous primitive which allows nested requests to the same resource, even when this is already owned by the requesting task.

In an `EventVar` object, many tasks can be blocked waiting for the signaling of an event. When its `Signal()` method is requested, all the blocked tasks are restarted.

Every synchronization object provides a method to test the blocking state of the object and a timeout argument in the blocking methods.

TasksMgr. The `TasksMgr` is the object that manages tasks. Its main functions are:

- Creation of new tasks. A method is provided to create additional tasks. The task manager needs only the agent and address where the new task has to be started and its stack size. Also, some arguments can be passed to the task in the `Create()` method call.
- Deletion of existing tasks. It can be explicitly requested or done implicitly. For the first case, we provide the `Delete()` method. The second case arises when the task has terminated the execution of the function or routine where it was started.
- Management of the threads belonging to each task. Threads are created when a task jumps to a different agent and they are deleted when a task returns. This is done automatically by the task manager, which is the only kernel object that knows about their existence. As a result, threads become transparent to the rest of kernel objects.

6.2 Implementation

Our kernel has been implemented using C++. Thus, all the objects shown in the previous sections are implemented as different C++ classes.

Special care has been taken to separate the class interface from its implementation. So, every kernel object is represented by an abstract class who contains only its interface. The whole set of abstract classes is an architecture-independent model of the kernel and it is the result of its design stage.

Using the inheritance capability of the C++ language, as recommended in [10], abstract classes can be refined to other implemented classes which adapt the architecture-independent interface provided by the abstract class to code that can be run in the target machine. Moreover, only a little part of the implemented classes deal with hardware details; the rest is highly portable.

Thus, each kernel object class belongs to one of the following groups:

- Classes which are implemented using architecture-independent code. Both abstract and implemented classes are directly portable to other architectures. Examples of this kind are the `TasksMgr` and the `ObjectMgr` classes.
- Classes which can be partially implemented using architecture-independent code. The methods of the abstract class are implemented using machine-independent C++ code. The specific details concerning the target machine are added in a subclass of this non-abstract class. Only this subclass will change when the kernel is ported to a different architecture.

For instance, the `PmegMgr` class is a subclass of the `PageTable` class. The latter maintains the page replacement algorithm and other machine-independent features related to virtual to physical address conversion, whilst the former has all the machine-dependent details.

- Classes whose code is architecture-dependent. Only the abstract class is directly portable; i.e., the class interface is architecture-independent. They have to be re-implemented when the system is ported. Examples: `MMU`, `CPUContext` and `CPU`.

6.3 Code Size

Table 1 sums up the figures about the kernel size. All the source code is fully commented. Removing comments, approximately a 50 % of the lines would be eliminated.

Concept	**Source** (Lines)	**Exec** (Bytes)
WHOLE KERNEL	19.636	46.493
LANGUAGE		
C++	16.787	39.413
Assembler	2.849	7.080
ARCHITECTURE DEPENDENCY		
Independent	14.785	35.577
Dependent	4.851	10.916

Table 1: Executable and source code sizes.

This table shows first the entire size of the kernel source code (in lines) and that of the kernel text image (in bytes).

The second group of figures divides the kernel code according to the implementation language. As we can see, only a 15% of the source code has been written in assembler. Once this code has been compiled, its 7.080 bytes mean the 15% of the total object code, too.

Considering hardware dependency, the 25% of the source code must be translated to port the kernel to a different architecture. In the current PC release, this source code generates the 23% of the text image.

7 Summary

The NanOS microkernel offers support for object-oriented services and applications. Its main services include object invocation, memory management and device driver support. NanOS also provides a reduced set of abstractions which comprises: objects, agents and tasks.

The kernel itself has been organized as a set of cooperative objects. Some of them offer public interfaces to the upper levels. Synchronization objects, memory providers and virtual spaces are three examples of kernel objects of this kind.

Portability and efficiency were key design objectives of this system. Currently, we have two releases of our kernel running in two different architectures. So, the first property has been accomplished. Object invocation is the most critical service provided and the microkernel spends 22 μs to do an inter-agent invocation and the corresponding return to the calling agent in a PC machine with a 90 MHz Intel Pentium processor[1].

We plan to build a distributed object-oriented operating system offering, among others, a UNIX interface to applications. The NanOS kernel is the base for this future system. It has to be extended with network support in the next stage of its development. At the same time, we are designing an ORB which will provide a good platform to develop distributed applications and servers. We plan to extend this ORB with additional services such as the

[1] This measurement corresponds to an object call made at user level, with four arguments whose size do not exceed 32 bytes. So, this implies four changes of privilege level and two address space changes.

support and management of replicated objects, which will provide the basis for highly available applications and servers.

References

[1] J.M.BERNABÉU-AUBÁN, P.W.HUTTO, Y.A.KHALIDI, M.AHAMAD, W.F.APPELBE, P.DASGUPTA, R.J.LEBLANC, U.RAMACHANDRAN: *The Architecture of Ra: A Kernel for Clouds.* In Proc. of the 22nd Annual Hawaii International Conference on System Sciences, Jan. 1.989.

[2] B.BERSHAD, C.CHAMBERS, S.EGGERS, C.MAEDA, D.MCNAMEE, P.PARDYAK, S.SAVAGE, E.G.SIRER: *SPIN - An Extensible Microkernel for Application-Specific Operating System Services*. In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 68-71.

[3] D.R.CHERITON, K.J.DUDA: *A Caching Model of Operating System Kernel Functionality.* In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 88-91.

[4] E.W.DIJKSTRA: *Cooperating Sequential Processes.* Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1.965.

Reprinted in F.GENIUS(editor): *Programming Languages.* Academic Press, London, 1.968, pgs. 43-112.

[5] D.R.ENGLER, M.F.KAASHOEK, J.W.O'TOOLE: *The Operating System Kernel as a Secure Programmable Machine.* In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 62-67.

[6] G.HAMILTON, P.KOUGIOURIS: *The Spring Nucleus: A Microkernel for Objects.* Proc. of the 1993 Summer Usenix conference, Cincinnati, June 1.993.

[7] Y.KHALIDI, M.NELSON: *The Spring Virtual Memory System.* Sun Microsystems Laboratories Technical Report SMLI-93-9, Mountain View (California), March 1.993, 23 pgs.

[8] OBJECT MANAGEMENT GROUP: *Common Object Request Broker Architecture and Specification.* OMG Document Number 91.12.1.

[9] U.RAMACHANDRAN, S.MENON, R.J.LEBLANC, Y.A.KHALIDI, P.W.HUTTO, P.DASGUPTA, J.M.BERNABÉU-AUBÁN, W.F.APPELBE, M.AHAMAD: *Clouds: Experiences in Building an Object Based Distributed Operating System.* Technical Report, Georgia Institute of Technology, Atlanta, GA, June 1.989.

[10] V.F.RUSSO: *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1.990, 165 pgs.

[11] T.STIEMERLING, A.WHITCROFT, T.WILKINSON, N.WILLIAMS, P.OSMON: *Evaluating MESHIX - A UNIX Compatible Micro-Kernel Operating System.* In Proc. of the Autumn 1.992 OpenForum Technical Conference, Utrecht, The Netherlands, Nov. 1.992, pp. 45-58.

[12] L.VAN DOORN, P.HOMBURG, A.S.TANENBAUM: *Paramecium: An Extensible Object-Based Kernel.* Technical Report, Vrije Universiteit, Amsterdam, The Netherlands, 1.995.